C++ now

2025

Building Streams Out of Hot Air

Coinductive Types in C++ Senders

Steve Downey

BUILDING SENDER STREAMS OUT OF SENDERS

We're going to build an async stream, an infinite list, out of nothing but senders.

BUILDING SENDER STREAMS OUT OF SENDERS

We're going to build an async stream, an infinite list, out of nothing but senders.

Along the way, we'll take a look at the last 80 years of computer science, including some areas of current active research.

BUILDING SENDER STREAMS OUT OF SENDERS

We're going to build an async stream, an infinite list, out of nothing but senders.

Along the way, we'll take a look at the last 80 years of computer science, including some areas of current active research.

Having a better idea of how senders, and functions like them, are grounded in theory gives us a better idea of how they can be used and where to look to borrow designs and insights.

We don't just borrow [syntax]; on occasion, [C++] has pursued other languages down alleyways to beat them unconscious and rifle their pockets for new [semantics].

—(with apologies to) James D. Nicoll

I'm going to show you even a more horrible thing, a definition of CONS in terms of nothing but air, hot air.

—Gerald Jay Sussman, Computational Objects

Video Lectures - 5B: Computational Objects

Speaker notes

C++ P2300 Senders are computations to be performed—"an object that describes work". (see Niebler et al. 2024, 4.3) Taking designs that suspend computation are a natural place to begin when planning senders. Models that do so in an otherwise strict evaluation environment can be easier to reason about for porting to C++ than from default lazy environments, such as Haskell.

By 'air' Sussman meant building a data structure out of nothing but higher order lambda expressions in `scheme`. The technique is also a common implementation technique for functional programming languages, and turns out to be one of the ways pattern matching is supported.

This can be implemented directly using C++ lambda, particularly easily now that recursive lambda is possible using `deducing this`. We start out by implementing `Either`, then `Pair`, `Maybe`, `Boolean`, and then see how recursive types can be made, such as a `cons list`. The core of the pattern is closely related to generalized `fold`, or `catamorphism`, and has deep connections with the `Visitor` pattern.

Changing perspective from inductive types, like `list` to coinductive, infinite, types, like `stream`, means looking at deconstruction, or observation, of `codata`. The pattern of dispatching to handlers remains the same, though, with some slight inversion.

This guides us to a concrete Sender design which can be implemented as concrete sender types, rather than wrapping unnamed higher order functions. This also helps avoid recursion in the type system, hiding the uninteresting intermediate types being sent. Also, this provides an excuse to demonstrate implementing a sender. There are not enough examples, but writing a sender is intended to be within the scope of work for an intermediate C++ developer.

At the end we will have an async queue and an async stream implemented using just senders.

LAMBDA EXPRESSIONS

- The core:

  x

  a variable.

  $\lambda\, x\, .\, M$

  a function of one variable with definition *M*.

  M N

  application of the function *M* to the argument *N*.

- Β-reduction:
- ((λ x . *M*) *N*) → (*M*[x:= /N/])

- Notational Sugar
- Multi-variable extension
  - $\lambda\ x\ y\ .\ M \leftrightarrow \lambda\ x\ .\ (\lambda\ y\ .\ M)$
  - or
  - $\lambda\ x\ y\ .\ M \leftrightarrow \lambda\ x\ .\ \lambda\ y\ .\ M$
- Currying
  - $(f\ x\ y\ z) \leftrightarrow (((f\ x)\ y)\ z)$

CLOSURES

Closure

A function that retains access to the names contained in the scope in which it was created.

HIGHER-ORDER FUNCTIONS

Functions that can take or return functions.

Those might be closures.

THINGS ARE WHAT THEY DO

EITHER EXAMPLE WITH TYPECLASS MAP

SHAPE OF TYPECLASS MAP

```cpp
template <typename E, typename L, typename R> struct EitherTypeclass {
        constexpr auto left(L) const -> E;
        constexpr auto right(R) const -> E;
        constexpr auto isLeft(E) const -> bool;
        constexpr auto isRight(E) const -> bool;
        constexpr auto fromLeft(E, L) const -> L;
        constexpr auto fromRight(E, L) const -> L;
        template <typename C>
        constexpr auto
        either(E e, invocable_r<C, L> auto, invocable_r<C, R> auto) const -> C;
};
```

EITHER TYPECLASS FOR `std::expected`

```
template <typename L, typename R>
struct EitherTypeclass<std::expected<L, R>, L, R> {
        using E = std::expected<L, R>;
        constexpr auto left(L l) const -> E { return l; }
        constexpr auto right(R r) const -> E { return std::unexpected{r}; }
        constexpr auto isLeft(E e) const -> bool { return e.has_value(); }
        constexpr auto isRight(E e) const -> bool { return not e.has_value(); }
        constexpr auto fromLeft(E e, L l) const -> L {
                return isLeft(e) ? e.value() : l;
        }
        constexpr auto fromRight(E e, R r) const -> R {
                return isRight(e) ? e.error() : r;
        }

        template <typename C>
        constexpr auto either(E e,
                              invocable_r<C, L> auto left,
                              invocable_r<C, R> auto right) const -> C {
                return isLeft(e) ? left(e.value()) : right(e.error());
        }
};
```

TEST FUNCTION AND CONSTRUCTION

```
template <typename Either,
          typename Left,
          typename Right,
          auto either_map = either_typeclass<Either, Left, Right>>
void test_function() {
        constexpr auto l = either_map.left(7);
        constexpr auto r = either_map.right(9.0);
        constexpr bool b1 = either_map.isLeft(l);
        constexpr bool b2 = either_map.isLeft(r);
        constexpr bool b3 = either_map.isRight(l);
        constexpr bool b4 = either_map.isRight(r);
        static_assert(b1 == true);
        static_assert(b2 == false);
        static_assert(b3 == false);
        static_assert(b4 == true);
```

fromLeft AND fromRight

```
constexpr int k1 = either_map.fromLeft(l, 11);
constexpr int k2 = either_map.fromLeft(r, 11);
static_assert(k1 == 7);
static_assert(k2 == 11);

constexpr double k3 = either_map.fromRight(l, 11);
constexpr double k4 = either_map.fromRight(r, 11);
static_assert(k3 == 11.0);
static_assert(k4 == 9.0);
```

CASE SWITCH

```
        constexpr auto match = [=](auto e) {
                return either_map.template either<double>(
                        e,
                        [](auto x) -> double { return 2 * x; },
                        [](auto x) -> double { return 3 * x; });
        };
        constexpr double d1 = match(l);
        constexpr double d2 = match(r);
        static_assert(d1 == 14.0);
        static_assert(d2 == 27.0);
}
```

CALLING THE TEST FUNCTION

```
int main() { test_function<std::expected<int, double>, int, double>(); }
```

PAIR EXAMPLE WITH TYPECLASS MAP

SHAPE OF TYPECLASS MAP

```
template <typename P, typename L, typename R> struct PairTypeclass {
        constexpr auto pair(L, R) const -> P;
        constexpr auto first(P) const -> L;
        constexpr auto second(P) const -> R;
        template <typename C>
        constexpr auto apply(P e, invocable_r<C, L, R> auto) const -> C;
};
```

PAIR TYPECLASS FOR `std::pair`

```
template <typename L, typename R> struct PairTypeclass<std::pair<L, R>, L, R> {
        using P = std::pair<L, R>;
        constexpr auto pair(L l, R r) const -> P { return {l, r}; }
        constexpr auto first(P p) const -> L { return p.first; }
        constexpr auto second(P p) const -> R { return p.second; }

        template <typename C>
        constexpr auto apply(P p, invocable_r<C, L, R> auto f) const -> C {
                return f(p.first, p.second);
        }
};
```

TEST FUNCTION AND CONSTRUCTION

```
template <typename Pair,
          typename Left,
          typename Right,
          auto pair_map = pair_typeclass<Pair, Left, Right>>
void test_function() {
        constexpr auto p1 = pair_map.pair(7, 9.0);
```

APPLY

```
        constexpr auto match = [=](auto p) -> double {
                return pair_map.template apply<double>(
                        p, [](auto x, auto y) -> double {
                                return 2 * x + 3 * y;
                        });
        };
        constexpr double d1 = match(p1);
        static_assert(d1 == 41.0);
}
```

CALLING THE TEST FUNCTION

```
int main() { test_function<std::pair<int, double>, int, double>(); }
```

IMPLEMENTING DATA WITH LAMBDA

CLOSURES AND PARTIAL APPLICATION

Closures mean we can hold on to values.

Partial Application means we can defer using the values.

1. λ x f. f x
2. (λ x f. f x) a → λ f. f a
3. ((λ x f. f x) a) g → g a

CONTINUATION PASSING STYLE

Pass functions to closures to defer what to do next.

Two main strategies for encoding:

Church
: the *folds* or *catamorphisms* for an ADT

Scott
: the *pattern matching* or *visitor* for an ADT

RECURSIVE VS. NON-RECURSIVE TYPES

For non-recursive types, these are the same.

Either, Pair, Maybe, Boolean are non-recursive.

List is recursive.

Either

DEFINITION

```
data  Either a b
  = Left a
  | Right b
```

CONSTRUCTION

$$\textit{left} = \lambda\, a \,.\, \lambda\, l\, r \,.\, l\, a$$

$$\textit{right} = \lambda\, b \,.\, \lambda\, l\, r \,.\, r\, b$$

```
inline constexpr auto left = [](auto a) {
        return [a](auto l) { return [l, a](auto _) { return l(a); }; };
};

inline constexpr auto right = [](auto b) {
        return [b](auto _) { return [b](auto r) { return r(b); }; };
};
```

CASE ANALYSIS

$$\textit{either} = \lambda\, l\; r\; e.\; e\; l\; r$$

```
inline constexpr auto either = [](auto l) {
        return [l](auto r) { return [l, r](auto e) { return e(l)(r); }; };
};
```

Pair

DEFINITION

```
data  Pair l r
  = Pair l r
```

CONSTRUCTION

$$pair = \lambda\, l\, r\, .\, \lambda\, p.\, p\, l\, r$$

```
inline constexpr auto pair = [](auto l, auto r) {
        return [l, r](auto p) { return p(l, r); };
};
```

OBSERVATION

$$fst = \lambda\, p\, .\, p\, (\lambda\, l\, r.\, l)$$

$$snd = \lambda\, p\, .\, p\, (\lambda\, l\, r.\, r)$$

```
inline constexpr auto fst = [](auto p) {
        return p([](auto l, auto r) { return l; });
};

inline constexpr auto snd = [](auto p) {
        return p([](auto l, auto r) { return r; });
};
```

Maybe

DEFINITION

```haskell
data Maybe a
  = Nothing
  | Just a
```

CONSTRUCTION

$$nothing = \lambda\,.\,\lambda\, n\,.\,\lambda\, j\,.\, n$$

$$just = \lambda\, x\,.\,\lambda\, n\,.\,\lambda\, j\,.\, j\, x$$

```
inline constexpr auto nothing = []() {
        return [](auto n) { return [n](auto _) { return n(); }; };
};

inline constexpr auto just = [](auto x) {
        return [x](auto _) { return [x](auto j) { return j(x); }; };
};
```

OBSERVATION

isNothing = λ m . m (λ . true) (λ . false)

isJust = λ m . m (λ . false) (λ . true)

```
inline constexpr auto isNothing = [](auto m) {
        return m([]() { return true; })([](auto _) { return false; });
};
inline constexpr auto isJust = [](auto m) {
        return m([]() { return false; })([](auto _) { return true; });
};
inline constexpr auto fromJust = [](auto m) {
        return m([]() { std::abort(); })([](auto x) { return x; });
};
```

CASE ANALYSIS

$$\textit{maybe} = \lambda\ n\ j\ m\ .\ m\ n\ j$$

```
inline constexpr auto maybe = [](auto d) {
        return [d](auto f) {
                return [d, f](auto m) { return m([d]() { return d; })(f); };
        };
};
```

List

```
data List a
  = Nil
  | Cons a (List a)
```

CONSTRUCTION

$$\mathit{nil} = \lambda\ n\ c\ .\ n$$

CHURCH ENCODING

$$\mathit{cons} = \lambda\ x\ xs\ .\ \lambda\ n\ c\ .\ c\ x\ (xs\ n\ c)$$

SCOTT ENCODING

$$\mathit{cons} = \lambda\ x\ xs\ .\ \lambda\ n\ c\ .\ c\ x\ xs$$

```cpp
inline constexpr auto Nil = [](auto nil, auto cons) { return nil(); };

inline constexpr auto Cons = [](auto x, auto xs) {
        return [x, xs](auto nil, auto cons) { return cons(x, xs); };
};
```

OBSERVERS

isNil = λ l . l (λ x xs . false) true

head = λ l . l (λ x xs . x) error

- Church
  - *length* = λ l . l (λ x xs . (+) xs) 0
  - *tail* = λ l c n . l (λ x xs g . g x (xs c)) (λ xs . n) (λ x xs . xs)

  You are not expected to understand that.

- Scott:
  - *length* = λ l . l (λ x xs . (+) (length xs)) 0
  - *tail* = λ l . l (λ x xs . xs) nil

C++ CODE FOR SCOTT LIST

```
inline constexpr auto isNil = [](auto l) -> bool {
        return l([]() { return true; }, [](auto x, auto xs) { return false; });
};

inline constexpr auto head = [](auto l) {
        return l([]() { std::abort(); }, [](auto x, auto xs) { return x; });
};

inline constexpr auto tail = [](auto l) {
        return l([]() { std::abort(); }, [](auto x, auto xs) { return xs; });
};

inline constexpr auto length = [](this const auto &self, auto l) {
        return l([]() { return 0; },
                 [self](auto x, auto xs) { return 1 + self(xs); });
};
```

Speaker notes

https://hackage.haskell.org/package/gulcii-0.3/src/doc/encoding.md

THE PATTERN(S)

NON-RECURSIVE TYPES

For a type T with constructors *A*, *B*, *C*, … using types a_1, a_2, a_3, …

```
data T a1 a2 a3 a4
  = A a1 a2
  | B a2 a3
  | C a4
```

CONVERT THE CONSTRUCTORS TO FUNCTIONS

```
data T a1 a2 a3 a4
  = A a1 a2
  | B a2 a3
  | C a4
```

- $A \equiv \lambda\ a_1\ a_2\ .\ \lambda\ f_1\ f_2\ f_3\ .\ f_1\ a_1\ a_2$
- $B \equiv \lambda\ a_2\ a_3\ .\ \lambda\ f_1\ f_2\ f_3\ .\ f_2\ a_2\ a_3$
- $C \equiv \lambda\ a_4\ .\ \lambda\ f_1\ f_2\ f_3\ .\ f_3\ a_4$

A FUNCTION TAKING A *T*

Defined by pattern matching:

- f (A x y) = body_A
- f (B y z) = body_B
- f (C w) = body_C

ENCODE THE FUNCTION

$$f \equiv \lambda\ t\ .\ t\ (\lambda\ a_1\ a_2\ .\ body_A)\ (\lambda\ a_2\ a_3\ .\ body_B)\ (\lambda\ a_4\ .\ body_C)$$

Where *t* is the result of one of the encoded constructors, such as:

$$A \equiv \lambda\ a_1\ a_2\ .\ \lambda\ f_1\ f_2\ f_3\ .\ f_1\ a_1\ a_2$$

A T is encoded as a function that takes functions for each of the constructors.

It dispatches to the function that corresponds to the constructor used.

This is how *Pattern Matching* works.

CHURCH ENCODING FOR RECURSIVE TYPES

A data type T with :

- constructors $C_1 \dots C_k$,
- where and the *arity* of the i^{th} constructor is $ar(i)$,
- and let $\vec{C}$ be a vector of all the constructors.

$$C_i \equiv \lambda\, x_1 \dots x_{ar(i)} \,.\, \lambda\, c_1 \dots c_k \,.\, c_i\ (x_1\ \vec{C}) \dots (x_{ar(i)}\ \vec{C})$$

SCOTT ENCODING FOR RECURSIVE TYPES

A data type T with:

- constructors $C_1 \ldots C_k$,
- where and the *arity* of the i^{th} constructor is $ar(i)$.

$C_i \equiv \lambda\ x_1 \ldots x_{ar(i)}\ .\ \lambda\ c_1 \ldots c_k\ .\ c_i\ x_1 \ldots x_{ar(i)}$

Recursive types are basically identical in the Scott encoding.

Speaker notes

CONNECTIONS

Folds

Church and Scott encodings of products are just *foldr*.

Catamorphisms

Folds for Sum types.

Visitor

The "Gang of Four" Vistor is the implementation of pattern matching.

Continuation Passing

All of the encodings take continuations for what to do. Moreover, Senders are an automation of Continuation Passing Style.

DATA AND CODATA

We can also define a type not in terms of how it is constructed but in terms of how it is deconstructed, or consumed.

For a type like *Pair*, we become concerned with *fst* and *snd* which deconstruct into the components, rather than *Pair a b*. For simple types, the perspectives are equally expressive.

For infinite types, the codata deconstructor perspective can be more expressive, and also analytically tractable.

Codata is "new" research from the 21st Century.

CONSTRUCTION VS. OBSERVATION

> *[S]witching focus from the way values are built, (i.e. introduced), to the way they are used, (i.e. eliminated).*

Paul Downen, Codata in Action

Speaker notes

STATE, BEHAVIOR, IDENTITY

- The hallmarks of objects in OOP are entities with
  - State
  - Behavior
  - Identity

  Objects change over time, do things, and are distinct from other instances.

  Very much unlike values.

REFERENCES

References can not be just constructed independently.

References must be *observed* and might change independently.

References are more like codata than data.

In particular this explains why a reference member in a `struct` is so problematic.

STREAMS

Streams are an archetypical codata type.

The only operation we have on a Stream is to deconstruct it into a value and a Stream.

- Always infinite
- No empty stream - non-constructable
- Defined by observation APIs

DEFINITIONS

```haskell
data Stream a = Stream
  { head :: () -> a
  , tail :: () -> Stream a
  }
```

head and *tail* are functions in this definition so it can be *strict*.

We can't make a Stream, but if we have one can split it into the head element and the rest of the Stream.

This is an *Abstract* Data Type.

CODATA EXTENSION

```
codata Stream a where
  { head :: Stream a -> a
  , tail :: Stream a -> Stream a
  }
```

ENCODING CODATA

We encode the observers, the *deconstructors*, or *eliminators*, instead of the *constructors*.

Those become the elements of the *Visitor* interface.

head = λ s . λ h t . h s.head() *tail* = λ s . λ h t . t s.tail()

IMPLEMENTING SENDERS

WHAT IS A SENDER?

A description of async work.

Senders "deliver" or "send" their result to a receiver.

COMPLETION SIGNATURES

They must advertise the signatures they may call on the reciever channels:

- set_value
- set_error
- set_stopped

APIS TO PROVIDE HOOKS FOR

execution::get_completion_signatures

Can the reciever handle what the sender wants to deliver?

execution::connect

Make the connection between the sender and the continuation the results are delivered throuhg.

OUT OF THE BOX

SENDER FACTORIES

`execution::just`

Lift a value into a sender.

`execution::read_env`

Read from the *Environment* and deliver that value.

`execution::schedule`

Empty start of a work graph.

SENDER ADAPTERS

`execution::then`

map, *transform*, *fmap*, etc – the Functor interface.

`execution::let_value`

bind, *and_then*, etc – the Monad interface.

`execution::on`

Switch scheduler.

`execution::when_all`

Join many senders.

The adapters `then` and `let_value` are necessary and sufficient.

Possibly not the most efficient.

SENDERS CAN BE USER CODE

Currently "expert-friendly."

Not intended to be "expert only."

CODE EXAMPLES

- Senders for:
  - Either
  - Pair
  - Stream

QUESTIONS?

Remember a question starts with:

QUESTIONS?

Remember a question starts with:

- who

QUESTIONS?

Remember a question starts with:

- what

QUESTIONS?

Remember a question starts with:

- when

QUESTIONS?

Remember a question starts with:

- where

QUESTIONS?

Remember a question starts with:

- how

QUESTIONS?

Remember a question starts with:

- why

QUESTIONS?

or

A propositional statement

a statement that has a truth value, either true or false, but not both.

QUESTIONS?

and goes up at the end.

QUESTIONS?

"More of a comment than a question ..."

Is a propositional statement, but hold them for a moment.

COMMENTS?

THANK YOU!

CODE TEST

```cpp
int main() {
    std::cout << "hello, world\n";
}
```