C++ now

Concept Maps using C++23 Library Tech

Indirection to APIs for a Concept

Steve Downey

2024

CONCEPT MAPS USING C++23 LIBRARY TECH

Steve Downey

ABSTRACT

C++0x Concepts had a feature `Concept Maps` that allowed a set of functions, types, and template definitions to be associated with a concept and the map to be specialized for types that meet the concept.

Speaker notes

ABSTRACT

C++0x Concepts had a feature `Concept Maps` that allowed a set of functions, types, and template definitions to be associated with a concept and the map to be specialized for types that meet the concept.

This allowed open extension of a concept.

ABSTRACT

C++0x Concepts had a feature `Concept Maps` that allowed a set of functions, types, and template definitions to be associated with a concept and the map to be specialized for types that meet the concept.

This allowed open extension of a concept.

A definition could be provided that allows an algorithm to operate in terms of the API a concept presents and the map would define how those operations are implemented for a particular type.

ABSTRACT

C++0x Concepts had a feature `Concept Maps` that allowed a set of functions, types, and template definitions to be associated with a concept and the map to be specialized for types that meet the concept.

This allowed open extension of a concept.

A definition could be provided that allows an algorithm to operate in terms of the API a concept presents and the map would define how those operations are implemented for a particular type.

- This is similar to how Haskell's `typeclass` works.

LOST WITH `Concepts-Lite`

Speaker notes

The feature was very general, and lost as part of the `Concepts-Lite` proposal that was eventually adopted.

This loss of a level of indirection means that the APIs for a concept must be implemented by those names for a type, even when those names are not particularly good choices in the natural domain of a type rather than in the domain as a concept.

The proliferation of `transform` functions for functorial `map` is such a problem.

It is also a problem when adapting types that are closed for extension or do not permit member functions.

WHY?

Speaker notes

WHY?

- Don't know if you should

WHY?

- Don't know if you should
- Need to know if you could first

ALTERNATIVES

Speaker notes

ALTERNATIVES

- Virtual Interface

ALTERNATIVES

- Virtual Interface
- Adapters

ALTERNATIVES

- Virtual Interface
- Adapters
- Collection of CPOs

HARD TO SUPPORT

Speaker notes

EXAMPLE FROM C++0X CONCEPTS

STUDENT RECORD

```
class student record {
public:
  string id;
  string name;
  string address;
  bool   id_equal(const student record&);
  bool   name_equal(const student record&);
  bool   address_equal(const student record&);
};
```

Speaker notes

EQUALITY COMPARABLE

```
concept_map EqualityComparable<student record>{
    bool operator==(const student record& a,
                    const student record& b){
        return a.id_equal(b);
}
};
```

Speaker notes

ALLOW ASSOCIATED TYPES

Very useful for pointers

```
concept_map BinaryFunction<int (*)(int, int), int, int>
{
    typedef int result_type;
};
```

Speaker notes

WHY DIDN'T WE GET THEM?

Let's not go there right now.

Speaker notes

STATE OF THE ART

RUST TRAITS

```rust
trait PartialEq {
    fn eq(&self, rhs: &Self) -> bool;

    fn ne(&self, rhs: &Self) -> bool {
        !self.eq(rhs)
    }
}
```

Speaker notes

C++ CPOS

SOME CONCEPTS AND TYPES

```
namespace N::hidden {
template <typename T>
concept has_eq = requires(T const& v) {
  { eq(v, v) } -> std::same_as<bool>;
};

struct eq_fn {
  template <has_eq T>
  constexpr bool operator()(T const& x,
                            T const& y) const {
    return eq(x, y);
  }
};

template <has_eq T>
constexpr bool ne(T const& x, T const& y) {
  return not eq(x, y);
}

template <typename T>
concept has_ne = requires(T const& v) {
  { ne(v, v) } -> std::same_as<bool>;
};

struct ne_fn {
  template <has_ne T>
  constexpr bool operator()(T const& x,
                            T const& y) const {
    return ne(x, y);
  }
};
} // namespace N::hidden
```

See Why tag_invoke is not the solution I want by Barry Revzin

https://brevzin.github.io/c++/2020/12/01/tag-invoke/

Speaker notes

C++ PARTIAL_EQUALITY

```
namespace N {
inline namespace function_objects {
inline constexpr hidden::eq_fn eq{};
inline constexpr hidden::ne_fn ne{};
} // namespace function_objects

template <typename T>
concept partial_equality
  requires(std::remove_reference_t<T> const& t)
{
  eq(t, t);
  ne(t, t);
};
} // namespace N
```

See Why tag_invoke is not the solution I want by Barry Revzin

https://brevzin.github.io/c++/2020/12/01/tag-invoke/

Speaker notes

REQUIREMENTS FOR SOLUTION

Speaker notes

REQUIREMENTS FOR SOLUTION

- Tied to the type system

REQUIREMENTS FOR SOLUTION

- Tied to the type system
- Automatable

REQUIREMENTS FOR SOLUTION

- Tied to the type system
- Automatable
- "zero" overhead

REQUIREMENTS FOR SOLUTION

- Tied to the type system
- Automatable
- "zero" overhead
    - no virtual calls

REQUIREMENTS FOR SOLUTION

- Tied to the type system
- Automatable
- "zero" overhead
    - no virtual calls
    - no type erasure

WHAT DOES TYPECLASS DO?

Speaker notes

WHAT DOES TYPECLASS DO?

Adds a record to the function that defines the operations for the type.

WHAT DOES TYPECLASS DO?

Adds a record to the function that defines the operations for the type.

Can we do that?

TYPE-BASED LOOKUP

Speaker notes

TYPE-BASED LOOKUP

Templates!

ADDITIONAL REQUIREMENTS

Speaker notes

ADDITIONAL REQUIREMENTS

Avoid ADL

ADDITIONAL REQUIREMENTS

Avoid ADL

Object Lookup rather than Overload Lookup

VARIABLE TEMPLATES

Speaker notes

VARIABLE TEMPLATES

Variable templates have become more powerful

VARIABLE TEMPLATES

Variable templates have become more powerful

We can have entirely distinct specializations

A STEP TOWARDS IMPLEMENTATION

```
template <class T>
concept partial_equality = requires(
    std::remove_reference_t<T> const& t) {
  {
    partial_eq<T>.eq(t, t)
  } -> std::same_as<bool>;
  {
    partial_eq<T>.ne(t, t)
  } -> std::same_as<bool>;
};
```

Speaker notes

partial_eq<T>

AN INLINE VARIABLE OBJECT

```
template<class T>
constexpr inline auto partial_eq = hidden::partial_eq_default;
```

Speaker notes

A DEFAULT IMPLEMENTATION

```
constexpr inline struct partial_eq_default_t {
  constexpr bool
  eq(has_eq auto const& rhs,
     has_eq auto const& lhs) const {
    return (rhs == lhs);
  }
  constexpr bool
  ne(has_eq auto const& rhs,
     has_eq auto const& lhs) const {
    return (lhs != rhs);
  }
} partial_eq_default;
```

Speaker notes

NEW has_eq

```
template <typename T>
concept has_eq = requires(T const& v) {
  { operator==(v, v) } -> std::same_as<bool>;
};
```

Speaker notes

WILL DO BETTER

Speaker notes

WILL DO BETTER

In a bit

MONOID

MONOID

A little more than you think.

MONOID

A little more than you think.

- A type

MONOID

A little more than you think.

- A type
- With an associative binary operation

MONOID

A little more than you think.

- A type
- With an associative binary operation
- Which is closed

MONOID

A little more than you think.

- A type
- With an associative binary operation
- Which is closed
- And has an identity element

MAYBE NOT A LOT MORE

Speaker notes

MATH

Speaker notes

MATH

- $\oplus : M \times M \to M$

MATH

- $\oplus : M \times M \to M$
- $x \oplus (y \oplus z) = (x \oplus y) \oplus z$

MATH

- $\oplus : M \times M \to M$
- $x \oplus (y \oplus z) = (x \oplus y) \oplus z$
- $1_M \in M$ such that $\forall m \in M : (1_M \oplus m) = m = (m \oplus 1_M)$

FUNCTION FORM

Speaker notes

FUNCTION FORM

- $f: M \times M \to M$

FUNCTION FORM

- $f: M \times M \to M$
- $f(x, f(y, z)) = f(f(x, y), z)$

FUNCTION FORM

- $f: M \times M \to M$
- $f(x, f(y, z)) = f(f(x, y), z)$
- $1_M \in M$ such that $\forall m \in M: f(1_M, m) = m = f(m, 1_M)$

FUNCTION FORM

- $f: M \times M \rightarrow M$
- $f(x, f(y, z)) = f(f(x, y), z)$
- $1_M \in M$ such that $\forall m \in M: f(1_M, m) = m = f(m, 1_M)$

The similarity to left and right fold is **NOT** an accident

CORE FUNCTIONS

Speaker notes

CORE FUNCTIONS

$$empty : m$$

CORE FUNCTIONS

$$\boldsymbol{empty : m}$$
$$empty = concat\,[]$$

CORE FUNCTIONS

$$\boldsymbol{empty} : \boldsymbol{m}$$

$$empty = concat\, []$$

$$\boldsymbol{concat} : [\boldsymbol{m}] \rightarrow \boldsymbol{m}$$

CORE FUNCTIONS

$$\boldsymbol{empty : m}$$

$$empty = concat\,[]$$

$$\boldsymbol{concat : [m] \rightarrow m}$$

$$fold\ append\ empty$$

CORE FUNCTIONS

$$\boldsymbol{empty : m}$$

$$empty = concat\,[]$$

$$\boldsymbol{concat : [m] \to m}$$

$$fold\ append\ empty$$

$$\boldsymbol{append : m \to m \to m}$$

CORE FUNCTIONS

$$\boldsymbol{empty : m}$$

$$empty = concat\, []$$

$$\boldsymbol{concat : [m] \rightarrow m}$$

$$fold\; append\; empty$$

$$\boldsymbol{append : m \rightarrow m \rightarrow m}$$

$$op$$

CORE FUNCTIONS

$$\boldsymbol{empty : m}$$

$$empty = concat\,[]$$

$$\boldsymbol{concat : [m] \to m}$$

$$fold\ append\ empty$$

$$\boldsymbol{append : m \to m \to m}$$

$$op$$

Note that it's self-referential

CORE FUNCTIONS

$$\boldsymbol{empty : m}$$

$$empty = concat\,[]$$

$$\boldsymbol{concat : [m] \to m}$$

$$fold\ append\ empty$$

$$\boldsymbol{append : m \to m \to m}$$

$$op$$

Note that it's self-referential

This is common

FROM HASKELL PRELUDE

```haskell
class Semigroup a => Monoid a where
  mempty :: a
  mempty = mconcat []

  mappend :: a -> a -> a
  mappend = (<>)

  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

Speaker notes

MINIMUM SET

Speaker notes

MINIMUM SET

empty | *concat*

IN C++

```
template <typename T, typename M>
concept MonoidRequirements =
    requires(T i) {
      { i.identity() } -> std::same_as<M>;
    }
    ||
    requires(T i, std::ranges::empty_view<M> r1) {
      { i.concat(r1) } -> std::same_as<M>;
    };
```

Speaker notes

I am ignoring all sorts of const volatile reference issues here.

IMPLEMENTING THE OTHER SIDE

THE MAP FOR A MONOID

```
template <class Impl>
  requires MonoidRequirements<
      Impl,
      typename Impl::value_type>
struct Monoid : protected Impl {
  auto identity(this auto&& self);

  template <typename Range>
  auto concat(this auto&& self, Range r);

  auto op(this auto&& self, auto a1, auto a2);
};
```

Speaker notes

empty is a terrible name, concat only a little better. empty becomes identity

identity

```
auto identity(this auto && self) {
    std::puts("Monoid::identity()");
    return self.concat(std::ranges::empty_view<typename Impl::value_type>{});
}
```

concat

```
template<typename Range>
auto concat(this auto&& self, Range r) {
     std::puts("Monoid::concat()");
     return std::ranges::fold_right(r,
                  self.identity(),
                  [&](auto m1, auto m2){return self.op(m1, m2);});
 }
```

op

```
auto op(this auto&& self, auto a1, auto a2) {
     std::puts("Monoid::op");
     return self.op(a1, a2);
 }
```

DEDUCING `this` AND CRTP

DEDUCING `this` AND CRTP

We'll see in a moment, but it's because we want to constraint the required implementation.

DEDUCING `this` AND CRTP

We'll see in a moment, but it's because we want to constraint the required implementation.

We want to use the derived version which has all of the operations.

Plus

```
template <typename M>
class Plus {
public:
  using value_type = M;
  auto identity(this auto&& self) -> M {
    std::puts("Plus::identity()");
    return M{0};
  }

  auto op(this auto&& self, auto s1, auto s2) -> M {
    std::puts("Plus::op()");
    return s1 + s2;
  }
};
```

PlusMonoidMap

```
template<typename M>
struct PlusMonoidMap : public Monoid<Plus<M>> {
    using Plus<M>::identity;
    using Plus<M>::op;
};
```

Speaker notes

Need to pull the operations from the Monoid instance into the Map, so we get the right ones being used by concat.

This might be simpler if we didn't allow choice of the basis operations, but that's also overly restrictive.

THE MAP INSTANCES

```
template<class T> auto monoid_concept_map = std::false_type{};

template<>
constexpr inline auto monoid_concept_map<int> = PlusMonoidMap<int>{};

template<>
constexpr inline auto monoid_concept_map<long> = PlusMonoidMap<long>{};

template<>
constexpr inline auto monoid_concept_map<char> = PlusMonoidMap<char>{};
```

CAN WE concat INSTEAD?

```
class StringMonoid {
public:
  using value_type = std::string;

  auto op(this auto&&, auto s1, auto s2) {
    std::puts("StringMonoid::op()");
    return s1 + s2;
  }

  template <typename Range>
  auto concat(this auto&& self, Range r) {
    std::puts("StringMonoid::concat()");
    return std::ranges::fold_right(
        r, std::string{}, [&](auto m1, auto m2) {
          return self.op(m1, m2);
        });
  }
};
```

Speaker notes

No, I'm not properly constraining Range here. No, I'm not actually recommending this as an implementation.

THE MAP AND INSTANCE

```
struct StringMonoidMap : public Monoid<StringMonoid> {
    using StringMonoid::op;
    using StringMonoid::concat;
};

template<>
constexpr inline auto monoid_concept_map<std::string> = StringMonoidMap{};
```

SOME SIMPLE USE

EXERCISE THE FUNCTIONS

```
template<typename P>
void testP()
{
    auto d1 = monoid_concept_map<P>;

    auto x = d1.identity();
    assert(P{} == x);

    auto sum = d1.op(x, P{1});
    assert(P{1} == sum);

    std::vector<P> v = {1,2,3,4};
    auto k = d1.concat(v);
    assert(k == 10);
}
```

SOME SIMPLE CASES

```
    std::cout << "\ntest int\n";
    testP<int>();

    std::cout << "\ntest long\n";
    testP<long>();

   std::cout << "\ntest char\n";
    testP<char>();
```

ON `std::string`

This will use the StringMonoid we defined a few moments ago.

```
auto d2 = monoid_concept_map<std::string>;

std::cout << "\ntest string\n";
auto x2 = d2.identity();
assert(std::string{} == x2);

auto sum2 = d2.op(x2, "1");
assert(std::string{"1"} == sum2);

std::vector<std::string> vs = {"1","2","3","4"};
auto k2 = d2.concat(vs);
assert(k2 == std::string{"1234"});
```

Note that the map type is mostly invisible.

RESULTS

TEST INT

```
Plus::identity()
Plus::op()
Monoid::concat()
Plus::identity()
Plus::op()
Plus::op()
Plus::op()
Plus::op()
```

TEST LONG

```
Plus::identity()
Plus::op()
Monoid::concat()
Plus::identity()
Plus::op()
Plus::op()
Plus::op()
Plus::op()
```

TEST CHAR

```
Plus::identity()
Plus::op()
Monoid::concat()
Plus::identity()
Plus::op()
Plus::op()
Plus::op()
Plus::op()
```

TEST STRING

```
Monoid::identity()
StringMonoid::concat()
StringMonoid::op()
StringMonoid::concat()
StringMonoid::op()
StringMonoid::op()
StringMonoid::op()
StringMonoid::op()
```

MONOID IN TREES

FOLDABLE GENERALIZES

FOLDABLE GENERALIZES

Folding is very much tied to Range like things.

FOLDABLE GENERALIZES

Folding is very much tied to Range like things.

It can, and has, been generalized to things that can be traversed.

FOLDABLE GENERALIZES

Folding is very much tied to Range like things.

It can, and has, been generalized to things that can be traversed.

`monoids` are still critical for Traversables.

SUMMARIZING DATA IN A TREE

SUMMARIZING DATA IN A TREE

If the summary type is monoidal, nodes can hold summaries of all the data below them.

fingertrees

`fingertrees`

Much of the flexibility of `fingertrees` comes from the monoidal tags.

`fingertrees`

Much of the flexibility of `fingertrees` comes from the monoidal tags.

They are also fairly complicated.

`fingertrees`

Much of the flexibility of `fingertrees` comes from the monoidal tags.

They are also fairly complicated.

Technique can be applied to other, simpler trees.

`fingertrees`

Much of the flexibility of `fingertrees` comes from the monoidal tags.

They are also fairly complicated.

Technique can be applied to other, simpler trees.

P3200 (eventually) ((C++29))

FRINGE-TREE

Simplified tree with data at the edges

CODE

Show the monoid-map branch of

steve-downey/fringetree.git

SUMMARY FOR CONCEPT MAPS

Tell you what I told you

- Variable templates for map lookup
- Named operations on the map object
- Open for extension
- Concept checkable implementations
- Decoupled map use and implementation

QUESTIONS?

Or comments

THANK YOU

Speaker notes