Meta++

Language Support for Advanced Generative Metaprogramming

Andrew Sutton | Wyatt Childers

LOCK3 SOFTWARE, LLC

WARNING

This talk contains language features being proposed for a far-distant version of C++

This is a work in progress

These design of these features is incomplet and potentially inkorrect

Generative programming

Common approaches

Preprocessors – generate code from program source

DSLs – generate code from external languages

Generative programming

Common approaches

- Preprocessors – generate code from program source
- DSLs – generate code from external languages

Downsides

- Maintaining separate tools & languages, more complex build rules

Metaprogramming

Typically we mean compile-time programming

- Used to compute values or types
- Observe properties of the elements in the program

Metaprogramming

Typically we mean compile-time programming

Used to compute values or types

Observe properties of the elements in the program

Generative metaprogramming emphasizes the ability to generate code

Generative features in C++

Other features that generate code:

C++98: Macros
C++98: Templates
C++17: Fold expressions
C++23?: Expansion statements
C++23?: Static reflection

Macros

Generates tokens, which can be used to generate arbitrary code

```
#define assert(E) if (!(E)) std::abort();
```

Powerful if used responsibly, very easy to abuse

Not actually part of C++, doesn't interact with the type system

Templates

Generates concrete definitions of functions, classes, and variables

```
template<totally_ordered T>
T min(T a, T b) {
  return b < a ? b : a;
}
```

Fold expressions

Generates iterated binary operations over parameter packs

```
template<typename... Ts>
bool all_args(Ts... args) {
  return (... && args);
}
```

Expansion statements

Generates a sequence of statements from an iterable or destructurable sequence

```
template<typename ...Ts>
min_arg(Ts ...args) {
  auto min = head(args...);
  template for (auto x : tail(args...))
    if (x < min)
      min = x;
  return min;
}
```

Static reflection

Reification generates types, values, expressions, and ids from reflection values

```
typename(reflexpr(int)) // generates int
namespace(reflexpr(std)) // generates std
template(reflexpr(std::pair)) // generates std::pair
valueof(reflexpr(main)) // generates a pointer to main
idexpr(reflexpr(main)) // generates the expression main
unqualid(reflexpr(main)) // generates the id ‘main’
...
```

What’s missing?

Can’t factor out and encapsulate commonly recurring declarative patterns

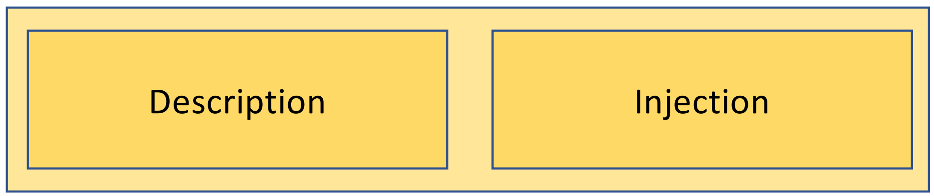
Would like to be able to generate parts of definitions

Assemble parts programmatically at compile-time

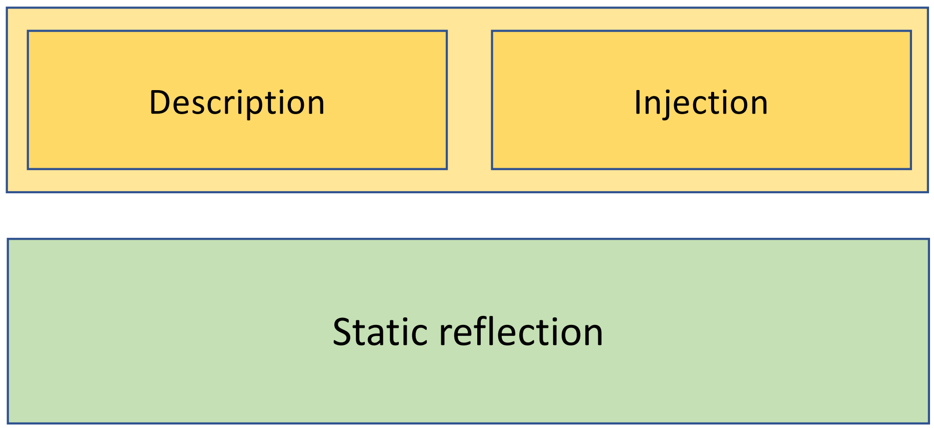
Features of generative metaprogramming

Code injection

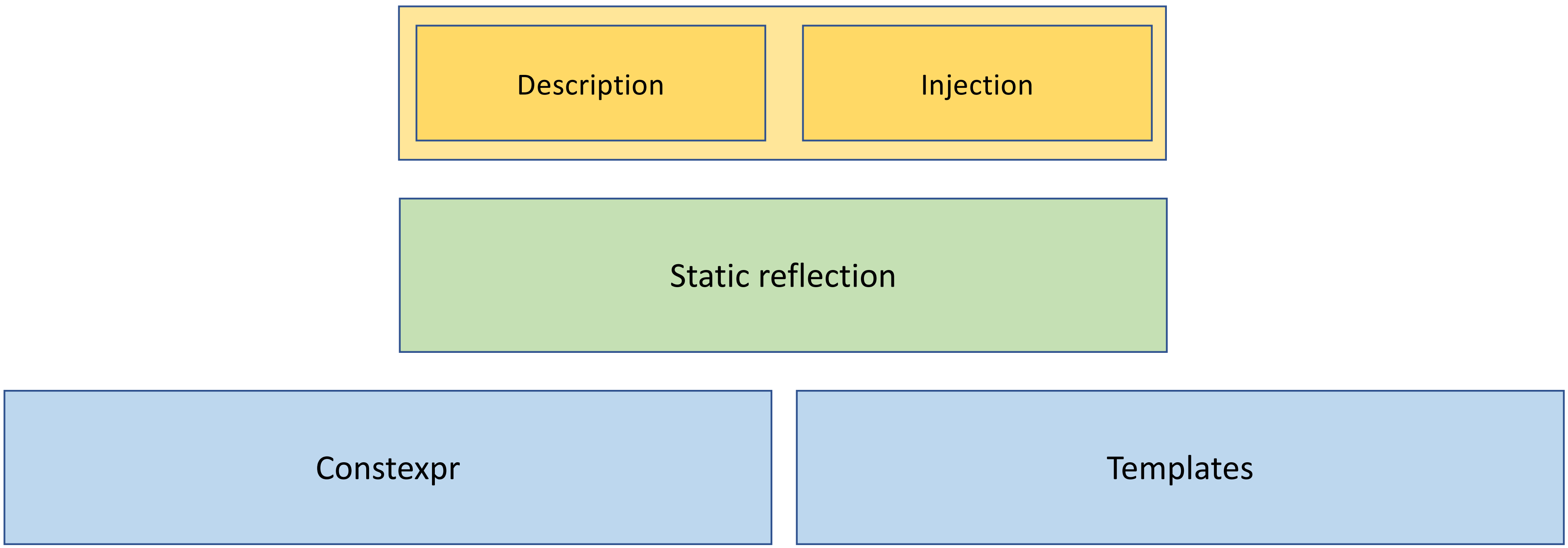
Features of generative metaprogramming



Features of generative metaprogramming



Features of generative metaprogramming



Examples

Examples are based on our Clang implementation

https://gitlab.com/lock3/clang

Documentation here:

https://gitlab.com/lock3/clang/wikis/home

Also: https://cppx.godbolt.org/

Example caveats

Examples may not compile as written

- Pushing some boundaries of what our compiler implements
- Cross your fingers, hope for the best

A first example

Extensible bitfields

We want inheritance-like composition of bitfields

- Used heavily in compilers to compress integral data in abstract syntax trees
- Example based on approach used in Clang

This is a slightly advanced entry point

- No simple examples

Extensible bitfields

```
struct expression_bits {
  unsigned value_category : 3;
  unsigned type_dependent : 1;
  unsigned value_dependent : 1;
  unsigned unexpanded : 1;
};
```

Extensible bitfields

```
struct fold_expression_bits : expression_bits {
  unsigned direction: 1;
};
```

Extensible bitfields

```
struct fold_expression_bits : expression_bits {
  unsigned direction: 1;
};
```

Extensible bitfields

```
struct fold_expression_bits {
  unsigned common_bits : 6;
  unsigned direction : 1;
};
```

Extensible bitfields

```
union expression_bitfields {
  expression_bits common;
  fold_expression_bits fold;
  unary_operator_bits unary;
  binary_operator_bits binary;
  // ...
};
```

Downsides of approach

```
struct fold_expression_bits {
  unsigned common_bits : 6;
  unsigned direction : 1;
};
```

Downsides of approach

```
struct fold_expression_bits {
  unsigned common_bits : 6;
  unsigned direction : 1;
};
```

Have to maintain size of common bits (easy with `constexpr`)

Can’t access common expression bits from this type

Extensible bitfields

```
struct fold_expression_bits {
  unsigned value_category : 3;
  unsigned type_dependent : 1;
  unsigned value_dependent : 1;
  unsigned unexpanded : 1;
  unsigned direction : 1;
};
```

Downsides of approach

```
struct fold_expression_bits {
  unsigned value_category : 3;
  unsigned type_dependent : 1;
  unsigned value_dependent : 1;
  unsigned unexpanded : 1;
  unsigned direction : 1;
};
```

Duplicate a lot of code (easily solved with macros)

Basic strategy

```
struct fold_expression_bits {
  // Insert common bits here
  unsigned direction : 1;
};
```

We need to describe what those common properties are

We need to inject them into the declaration

Class fragments

Describes a “part” of a class

```
constexpr auto common_bits = __fragment struct {
  unsigned value_category : 3;
  unsigned type_dependent : 1;
  unsigned value_dependent : 1;
  unsigned unexpanded : 1;
};
```

Class fragments

Describes a “part” of a class

```
constexpr auto common_bits = __fragment struct {
  unsigned value_category : 3;
  unsigned type_dependent : 1;
  unsigned value_dependent : 1;
  unsigned unexpanded : 1;
};
```

Class fragments

Describes a “part” of a class

```
constexpr auto common_bits = __fragment struct {
  unsigned value_category : 3;
  unsigned type_dependent : 1;
  unsigned value_dependent : 1;
  unsigned unexpanded : 1;
};
```

Source code injection

```
struct fold_expression_bits {
  consteval -> common_bits;
  unsigned direction : 1;
};
```

After injection

```
struct fold_expression_bits {
  unsigned value_category : 3;
  unsigned type_dependent : 1;
  unsigned value_dependent : 1;
  unsigned unexpanded : 1;
  unsigned direction : 1;
};
```

About injection

```
struct fold_expression_bits {
  consteval -> common_bits;
  unsigned direction : 1;
};
```

About injection

```
struct fold_expression_bits {
  consteval {
    -> common_bits;
  }
  unsigned direction : 1;
};
```

Metaprograms

```
struct fold_expression_bits {
  consteval {
    -> common_bits;
  }
  unsigned direction : 1;
};
```

Injection statement

```
struct fold_expression_bits {
  consteval {
    -> common_bits;
  }
  unsigned direction : 1;
};
```

Injection statement

```
struct fold_expression_bits {
  consteval {
    -> common_bits;
  } // <- Injection happens here
  unsigned direction : 1;
};
```

Façades

Iterator façade

Provides an interface that models concepts, while requiring only a handful of operations

```
x.deref()
x.incr()
x.equal(y)
```

Façade fragments

```
constexpr auto input_iterator_facade =
  __fragment struct iterator {
    // ...
  };
```

Façade fragments

```
constexpr auto input_iterator_facade =
  __fragment struct iterator {
    // ...
  };
```

Façade fragments

```
constexpr auto input_iterator_facade =
  __fragment struct iterator {
    decltype(auto) operator*() const {
      return this->deref();
    }
    // ...
  };
```

Façade fragments

```
constexpr auto input_iterator_facade =
  __fragment struct iterator {
    decltype(auto) operator*() const {
      return this->deref();
    }
    // ...
  };
```

Façade fragments

```
constexpr auto input_iterator_facade =
  __fragment struct iterator {
    // ...
    iterator& operator++() {
      this->incr();
      return *this;
    }
    // ...
  };
```

Façade fragments

```
constexpr auto input_iterator_facade =
  __fragment struct iterator {
    // ...
    iterator& operator++() {
      this->incr();
      return *this;
    }
    // ...
  };
```

Façade fragments

```
constexpr auto input_iterator_facade =
  __fragment struct iterator {
    // ...
    bool operator==(iterator const& x) const {
      return this->equal(x);
    }
   bool operator!=(iterator const& x) const {
      return !this->equal(x);
    }
  };
```

Façade fragments

```
constexpr auto input_iterator_facade =
  __fragment struct iterator {
    // ...
    bool operator==(iterator const& x) const {
      return this->equal(x);
    }
   bool operator!=(iterator const& x) const {
      return !this->equal(x);
    }
  };
```

Using façades

```
template<typename T>
struct list_iterator {
  consteval -> input_iterator_facade;

  T const& deref() const;
  void incr();
  bool equal(list_iterator const& x);

  list_node<T>* node;
};
```

Injection semantics

When injecting the name of fragment is substituted for the name of the enclosing class. This name:

```
__fragment struct iterator {
```

Is replaced by this name:

```
struct list_iterator {
```

Results of injection

```
template<typename T>
struct list_iterator {
  auto operator*() const { return this->deref(); }
  list_iterator& operator++() { this->incr(); return *this; }
  // ...

  T const& deref() const;
  void incr();
  // ...
```

Getters

A book with some properties

```
struct book {
  [[get]] std::string title;
  [[get]] std::string author;
  [[get]] int page_count;

  consteval {
    gen_getters(reflexpr(book));
  }
};
```

A book with some properties

```
struct book {
  [[get]] std::string title;
  [[get]] std::string author;
  [[get]] int page_count;

  consteval {
    gen_getters(reflexpr(book));
  }
};
```

A book with some properties

```
struct book {
  [[get]] std::string title;
  [[get]] std::string author;
  [[get]] int page_count;

  consteval {
    gen_getters(reflexpr(book));
  }
};
```

Generating getters

```
consteval void gen_getters(meta::info cls) {
  auto members = std::members_of(cls);
  for (meta::info member : members)
    if (meta::is_nonstatic_data_member(member))
      if (meta::has_attribute(member, “get”))
        gen_getter(member);
}
```

Generating getters

```
consteval void gen_getters(meta::info cls) {
  auto members = std::members_of(cls);
  for (meta::info member : members)
    if (meta::is_nonstatic_data_member(member))
      if (meta::has_attribute(member, “get”))
        gen_getter(member);
}
```

Generating properties

```
consteval void gen_getter(meta::info m) {
  -> __fragment struct {
    typename(meta::type_of(m)) const&
    unqualid("get_", meta::name_of(m))() {
      return unqualid(meta::name_of(m));
    }
  };
}
```

Generating properties

```
consteval void gen_getter(meta::info m) {
  -> __fragment struct {
    typename(meta::type_of(m)) const&
    unqualid("get_", meta::name_of(m))() {
      return unqualid(meta::name_of(m));
    }
  };
}
```

Generating properties

```
consteval void gen_getter(meta::info m) {
  -> __fragment struct {
    typename(meta::type_of(m)) const&
    unqualid("get_", meta::name_of(m))() {
      return unqualid(meta::name_of(m));
    }
  };
}
```

Generating properties

```
consteval void gen_getter(meta::info m) {
  -> __fragment struct {
    typename(meta::type_of(m)) const&
    unqualid("get_", meta::name_of(m))() {
      return unqualid(meta::name_of(m));
    }
  };
}
```

Generating properties

```
consteval void gen_getter(meta::info m) {
  -> __fragment struct {
    typename(meta::type_of(m)) const&
    unqualid("get_", meta::name_of(m))() {
      return unqualid(meta::name_of(m));
    }
  };
}
```

Generating properties

```
consteval void gen_getter(meta::info m) {
  -> __fragment struct {
    typename(meta::type_of(m)) const&
    unqualid("get_", meta::name_of(m))() {
      return unqualid(meta::name_of(m));
    }
  };
}
```

Parameterized fragments

Names in fragments that refer to local variables are “implicit parameters” of the fragment

- Replaced within the fragment by a constant expression placeholder

The “value” of a fragment is a pair of:

- Reflection of class
- Mapping of implicit parameters to their computed values

Processing fragments

1. During parsing: identify and declare constexpr placeholders

2. During evaluation: create a mapping from placeholders to their corresponding values

3. During injection: replace placeholders with their values

Generating properties

```
consteval void gen_getter(meta::info m) {
  -> __fragment struct {
    typename(meta::type_of(m)) const&
    unqualid("get_", meta::name_of(m))() {
      return unqualid(meta::name_of(m));
    }
  };
}
```

Generating properties

```
consteval void gen_getter(meta::info m) {
  -> __fragment struct {
    typename(meta::type_of(m)) const&
    unqualid("get_", meta::name_of(m))() {
      return unqualid(meta::name_of(m));
    }
  };
}
```

Generated code

```
struct book {
  [[get]] std::string title;
  // ...
  std::string const& get_title() const {
    return title;
  }
  // ...
};
```

Existential types

Existential types

AKA abstract data types: conceptually a pair comprised of an abstract interface and a concrete implementation

- Related: runtime concepts, virtual concepts, any types, type erasure

Slides based on Sy Brand's implementation here

- https://github.com/TartanLlama/typeclasses

Some producers

```
struct static_producer {
  int produce() { return 42; }
};
struct dynamic_producer {
  int i = 0;
  int produce() { return i++; }
};
```

I want a list that contains both static and dynamic producers

Solution 1

```
struct producer {
  virtual int produce() = 0;
};
struct static_producer : producer { ... };
struct dynamic_producer : producder { ... };

std::vector<std::unique_ptr<producer>> producers;
```

Solution 1

```
struct producer {
  virtual int produce() = 0;
};
struct static_producer : producer { ... };
struct dynamic_producer : producder { ... };

std::vector<std::unique_ptr<producer>> producers;
```

Solution 2

Use type erasure to create a value-semantic wrapper around internally managed producer objects

Solution 2

Use type erasure to create a value-semantic wrapper around internally managed producer objects

Implementing a type-erased data structure by hand is hard

Introducing metaclasses

```
class(existential) producer {
  int produce();
};

std::vector<producer> producers;
producers.emplace_back(static_producer{});
producers.emplace_back(dynamic_producer{});
```

Introducing metaclasses

```
class(existential) producer {
  int produce();
};
```

A metaclass is a metaprogram that generates a new class from a prototype definition

Actually just syntactic sugar for already discussed features

Introducing metaclasses

```
namespace __hidden {
  struct producer { int produce(); }
};

struct producer {
  using prototype = __hidden::prototype;
  consteval { existential(reflexpr(prototype)); }
};
```

Generating existential types

```
consteval void existential(meta::info proto) {
  -> __fragment class X {
  public:
    storage<X> storage_;
    X() = delete;
    template <typename U> X(U u) : storage_(std::move(u)) { }
  };
  generate_call_forwarders(proto);
}
```

Generating existential types

```
consteval void existential(meta::info proto) {
  -> __fragment class X {
  public:
    storage<X> storage_;
    X() = delete;
    template <typename U> X(U u) : storage_(std::move(u)) { }
  };
  generate_call_forwarders(proto);
}
```

Stored objects

```
template<typename Abstract>
struct storage {
  template <class Concrete>
  storage(Concrete&& x)
    : model(make_unique<impl<Abstract, Concrete>>(x))) {}

  unique_ptr<model<Abstract>> model;
};
```

Model and implementation

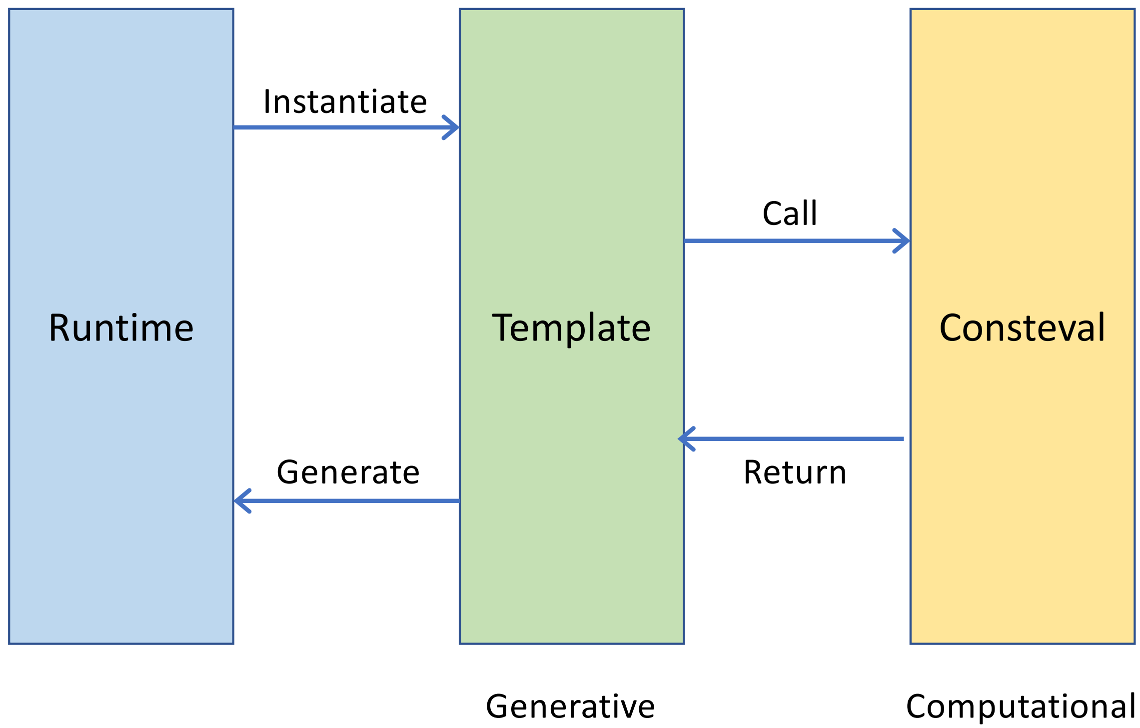
```
template<typename Abstract>
struct model {
  // ...
}

template<typename Abstract, typename Concrete>
struct impl : model<Abstract> {
  // ...
  Concrete obj;
};
```

Introducing metaclasses

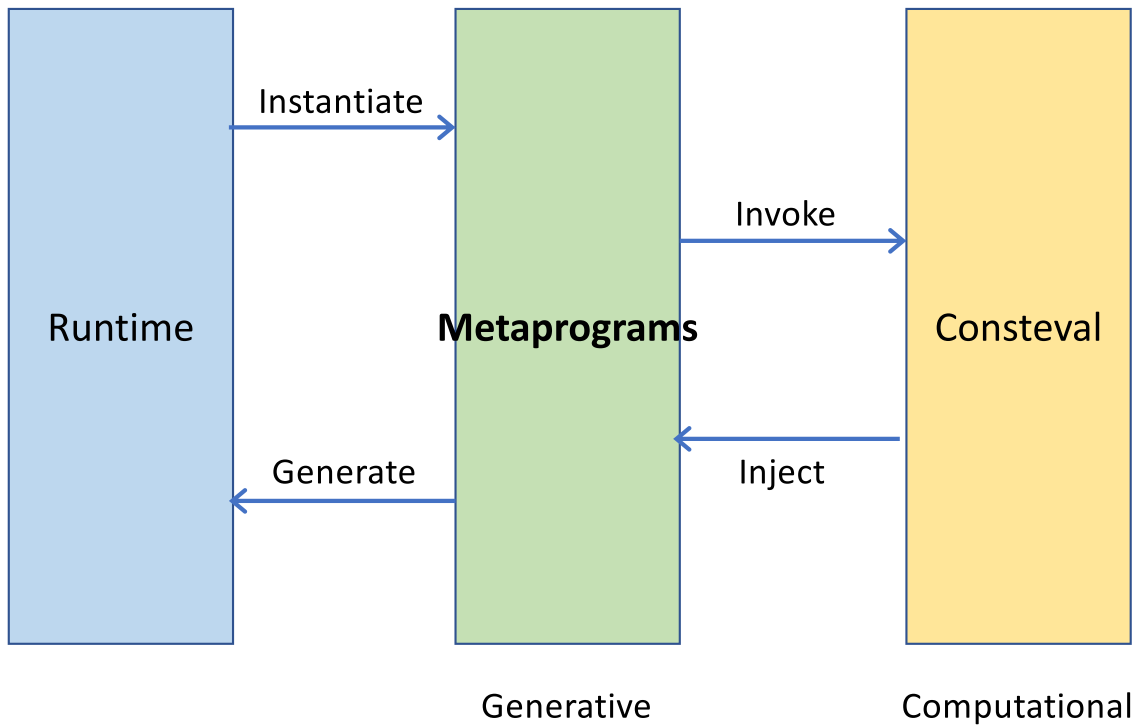
```
class(existential) producer {
  int produce();
};

std::vector<producer> producers;
producers.emplace_back(static_producer{});
producers.emplace_back(dynamic_producer{});
producers[0].produce(); // returns 42
```

Observations

Metaprogramming pattern



Metaprogram architecture



Final thoughts

Conclusions

Language support for Generative metaprogramming is a work in progress

I’ve only shown about a third of injection features

Appreciate feedback, suggestions, use cases, ideas

Thank you!

Questions?