Symbolic Calculus for High-performance Computing From Scratch Using C++23

VINCENT REVERDY

Cppcon
The C++ Conference
2023
October 01 - 06

Symbolic Calculus for High-Performance Computing from Scratch using C++23

Vincent Reverdy
Laboratoire d'Annecy de Physique des Particules, France

October 4th, 2023

Table of contents

Introduction

Starting from equations

$$\frac{d^2x}{dt^2} + 2\zeta w_0 \frac{dx}{dt} + w_0^2 x = 0$$

$$F = G\frac{m_1 m_2}{r^2}$$

$$y(t) = a \times \sin\left(\omega \times t + \varphi\right)$$

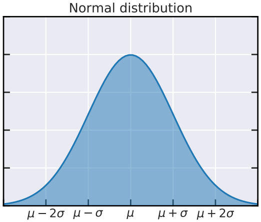
The topic of this talk

Wouldn't it be nice to be able to directly type and handle equations in C++ code?

Mathematical expressions

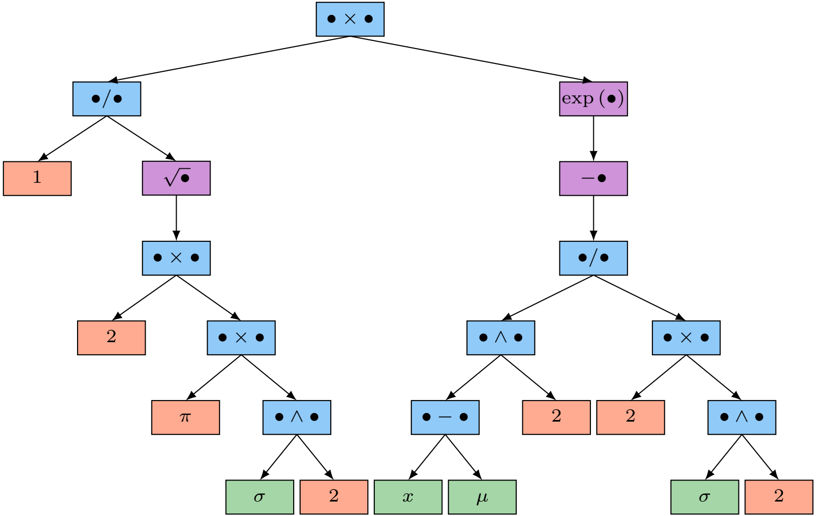
Normal distribution PDF

$$f = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$



Legend

Function (arity $= 2$): blue
Function (arity $= 1$): purple
Constants: orange
Variables: green



Preliminary note

Back in 2019

EDSL Infinity Wars: Mainstreaming Symbolic Computation, Joël Falcou and Vincent Reverdy, CppCon 2019

Hypothesis

This is the Scientific Computing Track so you all know about optimization, performance, parallelism, ...

What this talk is not about

- Complicated maths (you are smart people, you can do it yourself)
- High-performance computing (you all know about it + see the 2019 talk for that)
- Benchmarks, assembly, and optimization (see the 2019 talk for that) (and you are smart people, you can do it yourself)

What this talk is

A tutorial so you can build your own symbolic calculus tools from scratch in modern C++

The key idea

In 2019 we introduced...

Stateless expression templates

Old-school Expression Templates

- Data (vectors, matrices, tensors...) serve as terminal symbols
- Expression templates are stateful

Stateless Expression Templates

- Formulas are stateless
- Data is injected in formulas
- Both live their lives in different world

What changed since then?

C++20 and C++23: the idea remains the same, but it's possible to have even nicer interfaces easily

And most importantly

Design your abstractions from what you would like to type!

(or to say it differently, reverse-engineer the language to fit what you want to say!)

Typing equations

What we would like to type

```
symbol a;
symbol w;
symbol t;
symbol phi;

formula f = a * sin(w * t + phi);

double y = f(a = 5.0, w = 2.5, t = 1.6, phi = 0);
```

The lambda trick

The idea

Starting from the syntax

```
symbol a;
symbol w;
symbol t;
symbol phi;
```

The idea

a, ω, t, φ should all have a unique type

The problem

How to produce a different type everytime a new symbol is created?

The terrible approach

Maybe using macros?!?

The good approach

Doable in plain modern C++?

Playing with lambdas

A simple test with lambdas

```
auto a = []{};
auto b = []{};
auto c = []{};
auto d = []{};
```

The result

```
std::cout << std::is_same_v<decltype(a), decltype(a)> << std::endl; // 1
std::cout << std::is_same_v<decltype(a), decltype(b)> << std::endl; // 0
```

Conclusion

Each declaration of a lambda introduce a new type.

Mixing with lambdas and template parameter

Lambdas as default template parameter

```
template <class = decltype([]{})>
struct symbol {};
```

The result

```
symbol x;
symbol y;

std::cout << std::is_same_v<decltype(x), decltype(x)> << std::endl; // 1
std::cout << std::is_same_v<decltype(x), decltype(y)> << std::endl; // 0
```

Also works with NTTP

```
template <auto = []{}>
struct symbol {};
```

Conclusion

Mixing lambdas and default template parameter allow to create uniquely-typed symbols.

Using plain C++ to generate uniquely-typed symbols

The ONE trick to remember: the "*lambda*" trick

```
template <auto = []{}>
struct symbol {};

symbol a;
symbol w;
symbol t;
symbol phi;

std::cout << std::is_same_v<
      decltype(a),
      decltype(w)
> << std::endl; // 0
```

From the lambda trick to proper symbols

Desired mechanisms for working symbols

- Comparison of symbol types (useful for terms reordering)
- Binding mechanism between symbols and values
- Constraint and concept mechanism for symbols

Comparison

Hashing types to order them

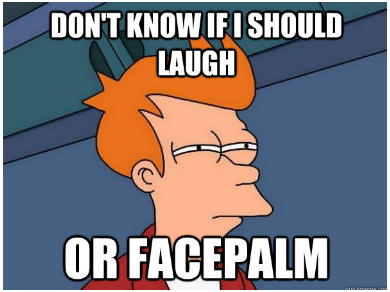
The obvious answer

Use `std::type_info::hash_code`

```
typeid(mysymbol).hash_code()
```

Except. . .

`std::type_info::hash_code` is still NOT `constexpr` in C++23



Introducing a symbol identifier

Wrapping the lambda in a symbol identifier

```
template <class>
struct symbol_id {};

template <auto Id = symbol_id<decltype([]{})>{}>
struct symbol {
    static constexpr auto id = Id;
};
```

Still working

```
symbol x;
symbol y;

std::cout << std::is_same_v<decltype(x), decltype(x)> << std::endl; // 1
std::cout << std::is_same_v<decltype(x), decltype(y)> << std::endl; // 0
```

Making the identifier comparable

Using a unique integer

Impossible as of C++23.

Using a pointer

```
template <class>
struct symbol_id {
    static constexpr auto singleton = []{};
    static constexpr const void* address = std::addressof(singleton);
};
```

The role of the singleton

The role of `singleton` is only to be something to get the address of.

The type of the singleton

`singleton` could be of any instantiable type (an integer for example).

Limitations of NTTP (1/2)

Non-Type Template Parameters (NTTP) (source: `cppreference`)

A non-type template parameter must have a structural type, which is one of the following types (optionally cv-qualified, the qualifiers are ignored):

- lvalue reference type (to object or to function)
- an integral type
- a pointer type (to object or to function)
- a pointer to member type (to member object or to member function)
- an enumeration type
- `std::nullptr_t` (since C++11)
- a floating-point type
- a literal class type with the following properties:
  - all base classes and non-static data members are public and non-mutable and
  - the types of all base classes and non-static data members are structural types or (possibly multi-dimensional) array thereof

Limitations of NTTP (2/2)

Symbol identifier final code

```
template <class>
struct symbol_id {
    static constexpr auto singleton = []{};
    static constexpr const void* address = std::addressof(singleton);
};

template <auto Id = symbol_id<decltype([]{})>{}>
struct symbol {
    static constexpr auto id = Id;
};
```

Consequence for symbol identifier

singleton and address have to stay public data members

Making symbol identifiers comparable

A naive strategy

```cpp
template <class Lhs, class Rhs>
constexpr bool operator<(symbol_id<Lhs>, symbol_id<Rhs>) {
    return symbol_id<Lhs>::address < symbol_id<Rhs>::address;
}
```

Pointer comparison

This is undefined according to the C++ standard if the pointers do not belong to the same array (which is the case here).

`std::less` does not have this limitation

```cpp
template <class Lhs, class Rhs>
constexpr bool operator<(symbol_id<Lhs>, symbol_id<Rhs>) {
    return std::less{}(symbol_id<Lhs>::address, symbol_id<Rhs>::address);
}
```

Leveraging the spaceship operator

Final code for comparison

```
template <class>
struct symbol_id {
    static constexpr auto singleton = []{};
    static constexpr const void* address = std::addressof(singleton);
};

template <class Lhs, class Rhs>
constexpr std::strong_ordering operator<=>(
    symbol_id<Lhs>,
    symbol_id<Rhs>
) {
    return std::compare_three_way{}(
        symbol_id<Lhs>::address,
        symbol_id<Rhs>::address
    );
}
```

Binding

Binding mechanism

Starting from the syntax

```
symbol a;
symbol w;
symbol t;
symbol phi;

formula f = a * sin(w * t + phi);

double y = f(a = 5.0, w = 2.5, t = 1.6, phi = 0); // Binding here
```

The idea

`operator=` should return an object that binds the value to the symbol

A simple binder class

A minimal example

```
template <class Symbol, class T>
struct symbol_binder {
    constexpr symbol_binder(Symbol, T x): value(x) {}
    static constexpr Symbol symbol = {};
    T value;
};
```

Returning the binder

```
template <auto = []{}>
struct symbol {
    template <class T>
    constexpr symbol_binder<symbol, T> operator=(T value) {
        return symbol_binder(*this, value);
    }
};
```

Limitation and strategy

Minimizing copies

The current version copies everything which might be a problem for vectors and matrices.

What we ideally would like

Input type	Storage type
`T&`	`const T&`
`const T&`	`const T&`
`volatile T&`	`const volatile T&`
`const volatile T&`	`const volatile T&`
`T&&`	`const T`
`const T&&`	`const T`
`volatile T&&`	`const volatile T`
`const volatile T&&`	`const volatile T`

Strategy

- Remove rvalue references (and rvalue references only)
- Add `const` to the non-referenced type

Removing specific types of references

Remove reference

The C++ standard only includes `std::remove_reference`.

Remove lvalue references

```
template <class T>
struct remove_lvalue_reference: std::type_identity<T> {};

template <class T>
requires std::is_lvalue_reference_v<T>
struct remove_lvalue_reference<T>: std::type_identity<std::remove_reference_t<T>> {};

template <class T>
using remove_lvalue_reference_t = remove_lvalue_reference<T>::type;
```

Remove rvalue references

```
template <class T>
struct remove_rvalue_reference: std::type_identity<T>{};

template <class T>
requires std::is_rvalue_reference_v<T>
struct remove_rvalue_reference<T>: std::type_identity<std::remove_reference_t<T>>{};

template <class T>
using remove_rvalue_reference_t = remove_rvalue_reference<T>::type;
```

Const-qualifying the non-referenced type

The problem

- `std::add_const_t<T&>` ⇒ `T&`
- We would like `f<T&>` ⇒ `const T&`

The solution

```
template <class T>
struct requalify_as_const: std::conditional<
    std::is_lvalue_reference_v<T>,
    std::add_lvalue_reference_t<std::add_const_t<std::remove_reference_t<T>>>,
    std::conditional_t<
        std::is_rvalue_reference_v<T>,
        std::add_rvalue_reference_t<std::add_const_t<std::remove_reference_t<T>>>,
        std::add_const_t<T>
    >
> {};
template <class T>
using requalify_as_const_t = requalify_as_const<T>::type;

template <class T>
struct requalify_as_volatile;
template <class T>
using requalify_as_volatile_t = requalify_as_volatile<T>::type;

template <class T>
struct requalify_as_cv;
template <class T>
using requalify_as_cv_t = requalify_as_cv<T>::type;
```

A copy-avoiding version of symbol binder

Final lightweight binder leveraging the previously-defined traits

```
// Symbol binder class definition
template <class Symbol, class T>
struct symbol_binder {
    // Types and constants
    using symbol_type = Symbol;
    using value_type = std::remove_cvref_t<T>;
    static constexpr Symbol symbol = {};
    // Constructors
    template <class U>
    requires std::is_convertible_v<U&&, requalify_as_const_t<remove_rvalue_reference_t<T>>>
    constexpr symbol_binder(Symbol, U&& x) noexcept(
        std::is_nothrow_convertible_v<U&&, requalify_as_const_t<remove_rvalue_reference_t<T>>>
    ): value(std::forward<U>(x)) {}
    // Accessors
    const value_type& operator()() const noexcept {return value;}
    // Implementation details: data members
    private:
    requalify_as_const_t<remove_rvalue_reference_t<T>> value;
};
// Deduction guide
template <class Symbol, class T>
symbol_binder(Symbol, T&&) -> symbol_binder<Symbol, T&&>;

// Symbol
template <auto = []{}>
struct symbol {
    template <class T>
    constexpr symbol_binder<symbol, T&&> operator=(T&& value) const {
        return symbol_binder(*this, std::forward<T>(value));
    }
};
```

Constraints

Symbols with constraints

It would be nice to be able to provide information on the mathematical entity it represents

```
symbol a;
symbol w;
symbol t;
symbol phi;
```

```
symbol<real> a;
symbol<real> w;
symbol<real> t;
symbol<real> phi;
```

Motivations

- Provide context for users
- Avoid unexpected conversions
- Leverage mathematical knowledge to optimize
- Mathematical operations behaviour may depend on mathematical concepts (commutativity for example)

The idea

Implement some kind of symbolic concepts.

Implementing a concept mechanism

Problem as of C++23

Concepts cannot be provided as template parameters

Lambda-based solution

```cpp
struct unconstrained {
    template <class T>
    constexpr std::true_type operator()(T x) const noexcept {return {};}
};

struct real {
    template <class T>
    constexpr std::false_type operator()(T x) const noexcept {return {};}
    template <class T>
    requires std::is_floating_point_v<T>
    constexpr std::true_type operator()(T x) const noexcept {return {};}
};

template <class T = unconstrained, auto = []{}>
struct symbol {
    template <class Arg>
    requires decltype(std::declval<T>()(std::declval<Arg>()))::value
    constexpr symbol_binder<symbol, Arg&&> operator=(Arg&& arg) const {
        return symbol_binder(*this, std::forward<Arg>(arg));
    }
};
```

An alternative approach

Trait-based solution

```cpp
template <class T>
struct unconstrained: std::true_type {};

template <class T>
struct real: std::is_floating_point<T> {};

template <template <class...> class Trait = unconstrained, auto = []{}>
struct symbol {
    template <class Arg>
    requires Trait<std::remove_cvref_t<Arg>>::value
    constexpr symbol_binder<symbol, Arg&&> operator=(Arg&& arg) const {
        return symbol_binder(*this, std::forward<Arg>(arg));
    }
};
```

Strong limitation of C++

C++ is currently type-generic but not kind-generic: template parameters kinds (types, NTTPs, template templates) cannot be mixed.

The absence of kind-genericity

The problem

The following objects cannot be specializations of the same template, cannot share the same identifier:

```
// Metafunction hierarchy
template<class T>
struct metafunction_wrapper_0 {};
template<template<class...> class F>
struct metafunction_wrapper_1 {};
template<template<template<class...> class...> class F>
struct metafunction_wrapper_2 {};
template<template<template<template<class...> class...> class...> class F>
struct metafunction_wrapper_3 {};
template<template<template<template<template<class...> class......> class...> class...> class F>
struct metafunction_wrapper_4 {};

// Use cases
metafunction_wrapper_1<metafunction_wrapper_0> x1; // OK
metafunction_wrapper_2<metafunction_wrapper_1> x2; // OK
metafunction_wrapper_3<metafunction_wrapper_2> x3; // OK
metafunction_wrapper_4<metafunction_wrapper_3> x4; // OK
```

A working approach

Wrapping traits in types

```
struct unconstrained {
    template <class T>
    struct trait: std::true_type {};
};

struct real {
    template <class T>
    struct trait: std::is_floating_point<T> {};
};

template <class Trait = unconstrained, auto = []{}>
struct symbol {
    template <class Arg>
    requires Trait::template trait<std::remove_cvref_t<Arg>>::value
    constexpr symbol_binder<symbol, Arg&&> operator=(Arg&& arg) const {
        return symbol_binder(*this, std::forward<Arg>(arg));
    }
};
```

Architecture

A symbol to rule them all

Reminder: desired mechanisms for working symbols

- Comparison of symbol types (useful for terms reordering)
- Binding mechanism between symbols and values
- Constraint and concept mechanism for symbols

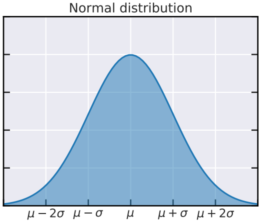
Combining the previous techniques

```
template <
    class Trait = unconstrained,
    auto Id = symbol_id<decltype([]{})>{}
> struct symbol {
    // Unique identifier
    static constexpr auto id = Id;
    // Binding mechanism
    template <class Arg>
    requires Trait::template trait<std::remove_cvref_t<Arg>>::value
    constexpr symbol_binder<symbol, Arg&&> operator=(Arg&& arg) const {
        return symbol_binder(*this, std::forward<Arg>(arg));
    }
};
```

Anatomy of a mathematical expression

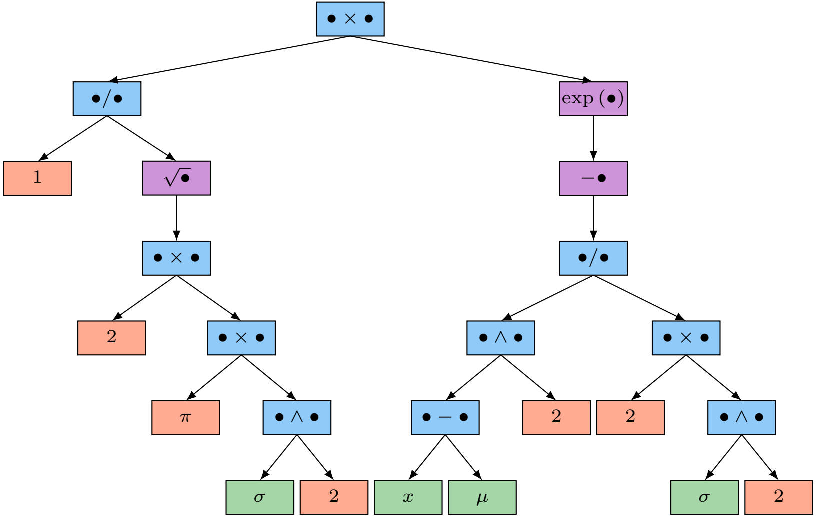
Normal distribution PDF

$$f = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$



Legend

Function (arity $= 2$): blue
Function (arity $= 1$): purple
Constants: orange
Variables: green



Concepts: the building blocks (symbols)

Symbols

- **variable symbols**: φ, ω, $\ldots$
- **constant symbols**: π, 5.6, 42 $\ldots$
- **function symbols**: $\sin$, $\log$, $\ldots$
  - **operator symbols**: $+$, $-$, $\times$

Free and bound variables

- **free variable**: variables that act as symbolic placeholders
- **bound variable**: variables that have been bound to a particular value

Concepts: the constructions (expressions)

$$y(t) = a \times \sin(\omega \times t + \varphi)$$

Symbolic expressions

- **term**: "a mathematical object": φ, ω, $\omega \times t + \varphi$, ...
- **formula**: "a mathematical sentence": $f = a \times \sin(\omega \times t + \varphi)$, ...
- **expression**: string of symbols
- **equation**: equality between two formulas: $a \times \sin(\omega \times t + \varphi) == x$

Additional concepts

- **subterm**: part of a term:
- **subexpression**: part of an expression
- **well-formed expression**: a string of symbols that satisfies the rule of the syntax
- **arity**: number of subterms: unary, binary, ternary, ...

Concepts: the actions (rewriting)

Symbolic rewriting

- **binding**: attaching a value to a symbolic variable
- **substitution**: replacing symbolic variables by their values in an expressions
- **rewriting**: replacing subterms by other terms in formulas

Abstract syntax trees

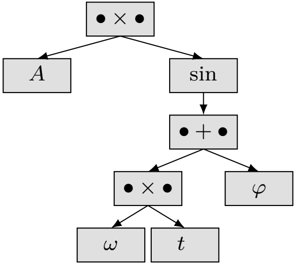
Formula example

$$f = a \times \sin\left(\omega \times t + \varphi\right)$$

Abstract syntax trees

Formula example

$$f = a \times \sin\left(\omega \times t + \varphi\right)$$



Abstract-Syntax Tree traversal

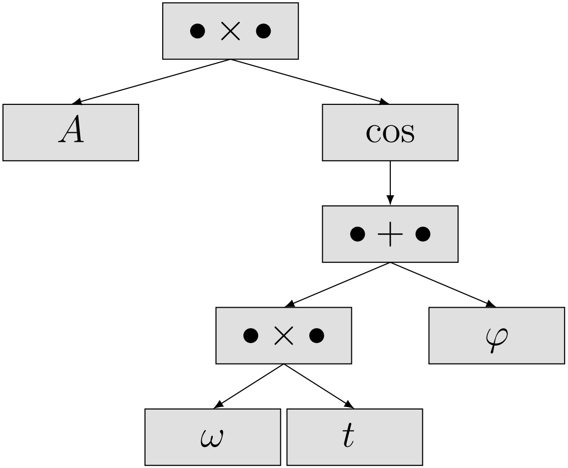
Mathematical expression

$$A \cos\left(\omega t + \varphi\right)$$

Abstract-Syntax Tree traversal

Mathematical expression

$$A \cos\left(\omega t + \varphi\right)$$



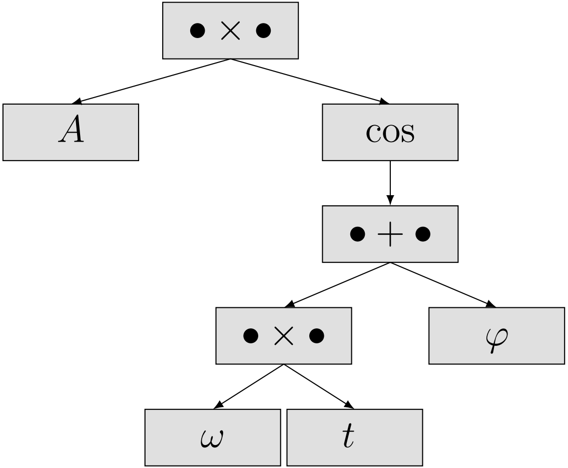
Abstract-Syntax Tree traversal

Mathematical expression

$$A \cos\left(\omega t + \varphi\right)$$

Pre-order

$$\times\left(A, \cos\left(+\left(\times\left(\omega, t\right), \varphi\right)\right)\right)$$



Abstract-Syntax Tree traversal

Mathematical expression

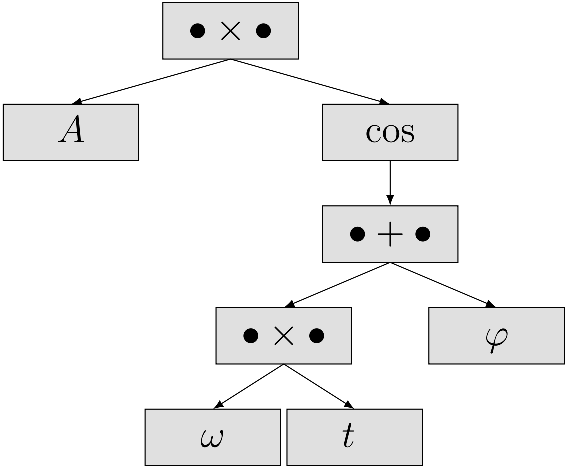
$$A\cos\left(\omega t+\varphi\right)$$

Pre-order

$$\times\left(A, \cos\left(+\left(\times\left(\omega, t\right), \varphi\right)\right)\right)$$

In-order

$$A\cos\left(\omega t+\varphi\right)$$



Abstract-Syntax Tree traversal

Mathematical expression

$$A\cos\left(\omega t+\varphi\right)$$

Pre-order

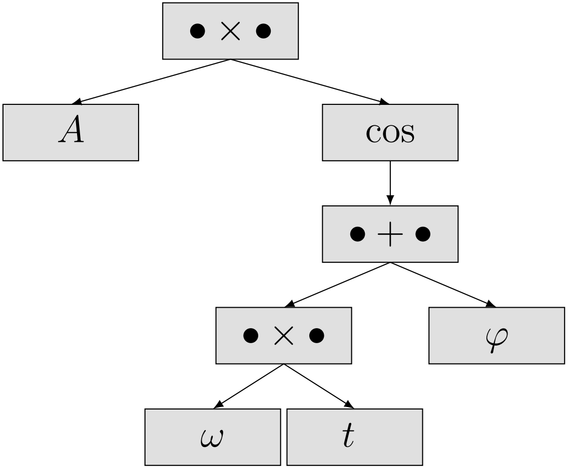
$$\times\left(A,\cos\left(+\left(\times\left(\omega,t\right),\varphi\right)\right)\right)$$

In-order

$$A\cos\left(\omega t+\varphi\right)$$

Post-order

$$A,\omega,t,\times,\varphi,+,\cos,\times$$



Starting with one concept

One concept to rule them all

```
// Type trait
template <class>
struct is_symbolic: std::false_type {};

// Variable template
template <class T>
inline constexpr bool is_symbolic_v = is_symbolic<T>::value;

// Concept
template <class T>
concept symbolic = is_symbolic_v<T>;
```

Example of specialization

```
// Specialization for variable symbol
template <class T, auto Id>
struct is_symbolic<symbol<T, Id>>: std::true_type {};
```

Substitution

Substitution in a formula

Introductory example

```
formula f = a * sin(w * t + phi);

double y = f(a = 5.0, w = 2.5, t = 1.6, phi = 0); // Substitution!
```

Starting from the formula

```
template <symbolic Expression>
struct formula {
    // Types and constants
    using expression = Expression;
    // Constructor
    constexpr formula(Expression expr) noexcept {};
    // Call operator where substitution happens
    template <class... Args>
    constexpr auto operator()(Args... args) const noexcept {
        return expression{}(substitution(args...));
    }
};
```

Overview

What a substitution class is

A pack of binders indexed by their identifier

General structure

```
// Index constant type
template <std::size_t I>
struct index_constant: std::integral_constant<std::size_t, I> {};

// Index constant variable template
template <std::size_t I>
inline constexpr index_constant<I> index = {};

// An indexed substitution element
template <std::size_t I, Binder B>
struct substitution_element;

// A helper class to build the substitution
template <class Sequence, class... T>
struct substitution_base;

// The substitution wrapper itself
template <class... Binders>
struct substitution;
```

Substitution as a pack of binders

Substitution

```cpp
// Substitution providing an index sequence to its base
template <class... Binders>
struct substitution: substitution_base<std::index_sequence_for<Binders...>, Binders...> {
    using base = substitution_base<std::index_sequence_for<Binders...>, Binders...>;
    using base::base;
    using base::operator[];
};

// Deduction guide
template <class... Binders>
substitution(const Binders&...) -> substitution<Binders...>;
```

Substitution base

```cpp
// Substitution base deriving from each binder
template <std::size_t... Index, class... Binders>
struct substitution_base<std::index_sequence<Index...>, Binders...>
: substitution_element<Index, Binders>... {
    using index_sequence = std::index_sequence<Index...>;
    using substitution_element<Index, Binders>::operator[]...;
    constexpr substitution_base(const Binders&... x): substitution_element<Index, Binders>(x)... {}
};
```

Substitution element

The magic of symbolic identifier indexing

```
template <std::size_t I, Binder B>
struct substitution_element {
    // Types and constants
    using index = index_constant<I>;
    using id_type = decltype(B::symbol_type::id);
    // Constuctor
    constexpr substitution_element(const Binder& b): _binder(b) {}
    // Access by index
    constexpr const T& operator[](index) const {
        return _binder;
    }
    // Access by id: all the magic happen here
    constexpr const T& operator[](id_type) const {
        return _binder;
    }
    // Implementation details: data members
    private: const B _binder;
};
```

Summary of the general mechanism

Example

```
symbol x;
symbol y;
s = substitution(x = 5, y = 2);
std::cout << s[x.id] << std::endl; // 5
std::cout << s[y.id] << std::endl; // 2
```

Step-by-step detail

```
// Step 1: substitution
template <class... Binders>
struct substitution: substitution_base<std::index_sequence_for<Binders...>, Binders...> {/*...*/};

// Step 2: substitution base
template <std::size_t... Index, class... Binders>
struct substitution_base<std::index_sequence<Index...>, Binders...>
: substitution_element<Index, Binders>... {/*...*/};

// Step 3: substitution element
template <std::size_t I, Binder B>
struct substitution_element {
    /*...*/
    constexpr const T& operator[](id_type) const {
        return _binder;
    }
};
```

From formulas to expressions

Going back to the formula

```
template <symbolic Expression>
struct formula {
    // Types and constants
    using expression = Expression;
    // Constructor
    constexpr formula(Expression expr) noexcept {};
    // Call operator where substitution happens
    template <class... Args>
    constexpr auto operator()(Args... args) const noexcept {
        return expression{}(substitution(args...));
    }
};
```

Example (not working at this point of the talk)

```
formula f = a * sin(w * t + phi);
double y = f(a = 5.0, w = 2.5, t = 1.6, phi = 0);
```

Last step

Building mathematical expressions

Construction

Introducing constant symbols

One little missing thing

```cpp
template <auto Value>
struct constant_symbol {
    using type = decltype(Value);
    static constexpr type value = Value;
}

// Making it symbolic
template <auto Value>
struct is_symbolic<constant_symbol<Value>>: std::true_type {};
```

Building expressions

Note

Now that all the pieces of the puzzle are here, things are not that difficult

Expressions

```
// The class for symbolic expressions
template <class Operator, symbolic... Terms>
struct symbolic_expression {};

// Making it symbolic
template <class Operator, symbolic... Terms>
struct is_symbolic<symbolic_expression<Operator, Terms...>>: std::true_type {};
```

Example of operator

```
template <symbolic Lhs, symbolic Rhs>
constexpr symbolic_expression<std::plus<void>, Lhs, Rhs> operator+(Lhs, Rhs) {return {};}
```

Overloading operators

Operators

```cpp
template <symbolic Lhs, symbolic Rhs>
constexpr symbolic_expression<std::plus<void>, Lhs, Rhs> operator+(Lhs, Rhs) noexcept {return {};}
template <symbolic Lhs, symbolic Rhs>
constexpr symbolic_expression<std::minus<void>, Lhs, Rhs> operator-(Lhs, Rhs) noexcept {return {};}
template <symbolic Lhs, symbolic Rhs>
constexpr symbolic_expression<std::multiplies<void>, Lhs, Rhs> operator*(Lhs, Rhs) noexcept {return {};}
template <symbolic Lhs, symbolic Rhs>
constexpr symbolic_expression<std::divides<void>, Lhs, Rhs> operator/(Lhs, Rhs) noexcept {return {};}
```

Custom function

```cpp
// Define function object
struct sin_symbol {
    template <class Arg>
    constexpr auto operator()(Arg&& arg) {
        return std::sin(std::forward<Arg>(arg));
    }
};

// Function builder
template <symbolic Arg>
constexpr symbolic_expression<sin_symbol, Arg> sin(Arg) noexcept {return {};}
```

Going further

At this point building formulas works

```
symbol a;
symbol w;
symbol t
symbol phi;
formula f = a * sin(w * t + phi);
```

What does not work yet

```
double y = f(a = 5.0, w = 2.5, t = 1.6, phi = 0);
```

Bringing back substitution

For expressions

```
template <class Operator, symbolic... Terms> struct symbolic_expression {
    template <class... Binders>
    constexpr auto operator()(const substitution<Binders...> &s) const noexcept {
        return Operator{}(Terms{}(s)...); // <— Everything happens here
    }
};
```

For symbols

```
template <class Trait = unconstrained, auto Id = symbol_id<decltype([]{})>{}> struct symbol {
    /*...*/
    template <class... Binders>
    constexpr auto operator()(const substitution<Binders...> &s) const {
        return s[id](); // <— Everything happens here
    }
};
```

For constants

```
template <auto Value> struct constant_symbol {
    /*...*/
    template <class... Binders>
    constexpr type operator()(const substitution<Binders...> &s) const {
        return value; // <— Everything happens here
    }
};
```

And finally. . .

. . . it works

```
// Program body
int main(int argc, char* argv[]) {

    // Defining mathematical symbols
    symbol a;
    symbol w;
    symbol t;
    symbol phi;

    // Writing a formula
    formula f = a * sin(w * t + phi);

    // Computing the result: the following prints -3.78401
    std::cout << f(a = 5.0, w = 2.5, t = 1.6, phi = 0) << std::endl;
}
```

Wait what, that's all?

Building and executing the AST is not that complicated once all the pieces are there.

Conclusion

In fact that's not all (exercices left to the user. . . or for another year)

Partial substitution

- For now only full substitution works
- Partial substitution need more work: `f(a = 5.0, w = 2.5)` should generate a new formula

Rewriting

- Replacing terms by others (rewriting) need also more work: `f(x = y / z)` should generate a new formula
- Simplification based on mathematical concepts
- Symbolic calculus (derivatives, integrals)
- Full blown custom rule-based rewriting

High-performance

- Since formulas have the entire information on the mathematical AST, it's possible to generate different tasks for calculations (large matrix multiplications on GPUs, complicated functions on CPUs, etc. . .)

And many more. . .

The sky is the limit

Summary

Key idea

- **Stateless expression templates**: formulas are stateless, and data is injected afterwards

Key trick

- **The lambda trick:** generate a new type for each declaration using lambda and default template parameter

Summary

- **symbolic**: the concept of every part of a symbolic expression
- **symbol**: to declare symbolic variable
- **symbol id**: to uniquely index symbols and make them comparable
- **symbol binder**: to bind a free symbol to a value
- **symbol constraint**: a concept wrapper to constrain symbols
- **constant symbol**: symbolic constants like π
- **symbolic expression**: the symbolic abstract syntax tree built from operators
- **formula**: to give name and call symbolic expressions
- **substitution**: the utility to replace symbols by their values and compute the result

And most importantly

Design your abstractions from what you would like to type!

(or to say it differently, reverse-engineer the language to fit what you want to say!)

Thank you for your attention

Any question?