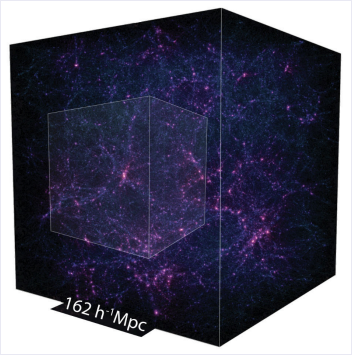
EDSL Infinity Wars
Mainstreaming Symbolic Computation

Joël Falcou & Vincent Reverdy

September 17th, 2019

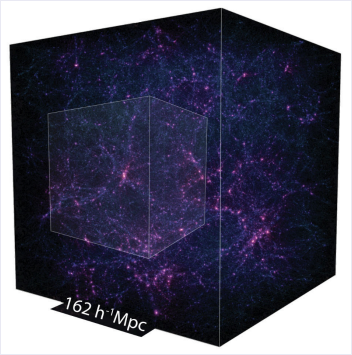
A story of numerical simulations

Numerical astrophysics

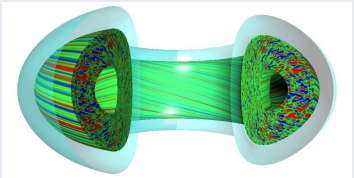


A story of numerical simulations

Numerical astrophysics

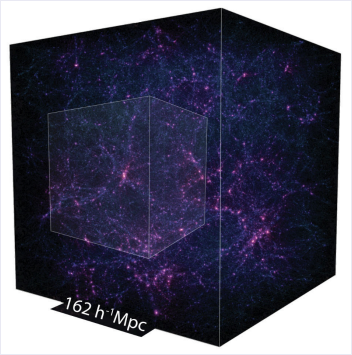


Magnetohydrodynamics simulations

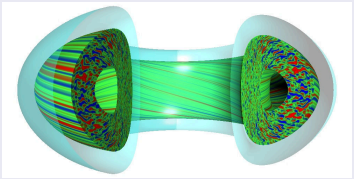


A story of numerical simulations

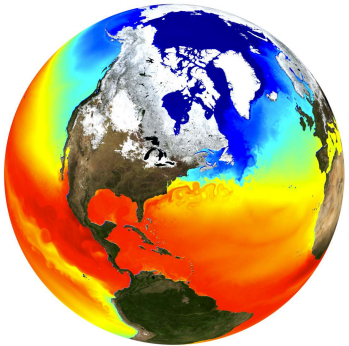
Numerical astrophysics



Magnetohydrodynamics simulations



Climate modeling



Simulations versus theory: Expectations vs Reality

Simulations versus theory: Expectations vs Reality

Simulations expectations

Theory expectations

Simulations versus theory: Expectations vs Reality

Simulations expectations

Theory expectations

Simulations reality

Theory Reality

Simulations versus theory

Simulations

Theory

Simulations versus theory

Simulations

- Partial Differential Equations (PDEs)

Theory

- Ordinary differential equations (ODEs)

Simulations versus theory

Simulations

- Partial Differential Equations (PDEs)
- Numerical methods

Theory

- Ordinary differential equations (ODEs)
- Symbolic calculus

Simulations versus theory

Simulations

- Partial Differential Equations (PDEs)
- Numerical methods
- Supercomputer

Theory

- Ordinary differential equations (ODEs)
- Symbolic calculus
- Laptop

Simulations versus theory

Simulations

- Partial Differential Equations (PDEs)
- Numerical methods
- Supercomputer
- General Purpose Languages (C++, C, Fortran...)

Theory

- Ordinary differential equations (ODEs)
- Symbolic calculus
- Laptop
- Domain Specific Languages (Mathematica...)

Simulations versus theory

Simulations

- Partial Differential Equations (PDEs)
- Numerical methods
- Supercomputer
- General Purpose Languages (C++, C, Fortran...)

Theory

- Ordinary differential equations (ODEs)
- Symbolic calculus
- Laptop
- Domain Specific Languages (Mathematica...)

Current status

Two different communities, two different ecosystems, two different set of tools...

Simulations versus theory

Simulations

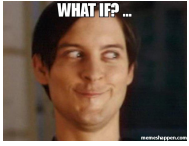
- Partial Differential Equations (PDEs)
- Numerical methods
- Supercomputer
- General Purpose Languages (C++, C, Fortran...)

Theory

- Ordinary differential equations (ODEs)
- Symbolic calculus
- Laptop
- Domain Specific Languages (Mathematica...)

Current status

Two different communities, two different ecosystems, two different set of tools...



Simulations versus theory

Simulations

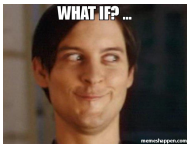
- Partial Differential Equations (PDEs)
- Numerical methods
- Supercomputer
- General Purpose Languages (C++, C, Fortran...)

Theory

- Ordinary differential equations (ODEs)
- Symbolic calculus
- Laptop
- Domain Specific Languages (Mathematica...)

Current status

Two different communities, two different ecosystems, two different set of tools...



Simulations versus theory

Simulations

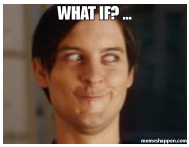
- Partial Differential Equations (PDEs)
- Numerical methods
- Supercomputer
- General Purpose Languages (C++, C, Fortran...)

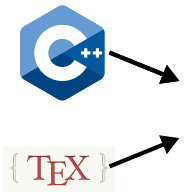
Theory

- Ordinary differential equations (ODEs)
- Symbolic calculus
- Laptop
- Domain Specific Languages (Mathematica...)

Current status

Two different communities, two different ecosystems, two different set of tools...





Simulations versus theory

Simulations

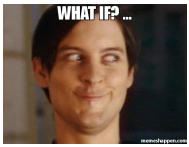
- Partial Differential Equations (PDEs)
- Numerical methods
- Supercomputer
- General Purpose Languages (C++, C, Fortran...)

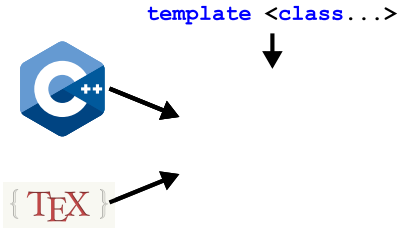
Theory

- Ordinary differential equations (ODEs)
- Symbolic calculus
- Laptop
- Domain Specific Languages (Mathematica...)

Current status

Two different communities, two different ecosystems, two different set of tools...





Simulations versus theory

Simulations

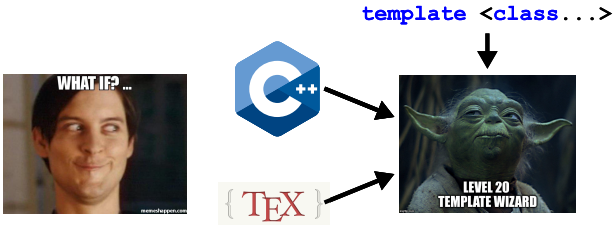
- Partial Differential Equations (PDEs)
- Numerical methods
- Supercomputer
- General Purpose Languages (C++, C, Fortran...)

Theory

- Ordinary differential equations (ODEs)
- Symbolic calculus
- Laptop
- Domain Specific Languages (Mathematica...)

Current status

Two different communities, two different ecosystems, two different set of tools...



Simulations versus theory

Simulations

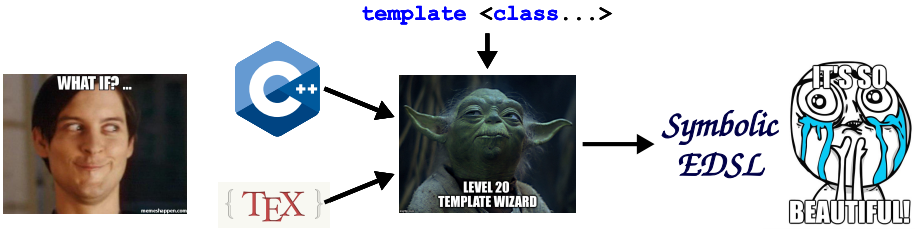
- Partial Differential Equations (PDEs)
- Numerical methods
- Supercomputer
- General Purpose Languages (C++, C, Fortran...)

Theory

- Ordinary differential equations (ODEs)
- Symbolic calculus
- Laptop
- Domain Specific Languages (Mathematica...)

Current status

Two different communities, two different ecosystems, two different set of tools...



EDSLs in a nutshell

Domain Specific Languages

- Non-Turing complete declarative languages
- Solve a single type of problems
- Express what to do instead of how to do it
- E.g: SQL, Make, Matlab, ...

EDSLs in a nutshell

Domain Specific Languages

- Non-Turing complete declarative languages
- Solve a single type of problems
- Express what to do instead of how to do it
- E.g: SQL, MAKE, MATLAB, ...

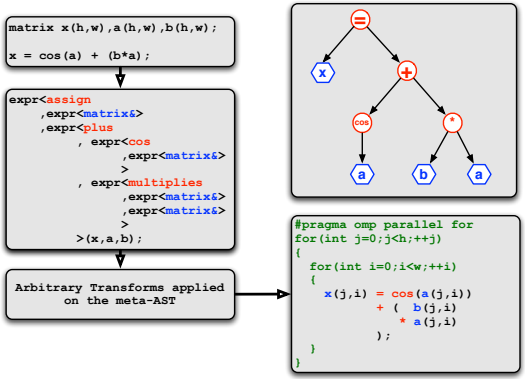
Embedded Domain Specific Languages [Abrahams 2004]

- A DSL incorporates domain-specific notation, constructs, and abstractions as fundamental design considerations.
- A Embedded Domain Specific Languages (EDSL) is simply a library that meets the same criteria
- Using Expression Templates is one way to design such libraries

The Expression Templates Idiom

Principles

- Relies on extensive operator overloading
- Carries semantic information around code fragment
- Introduces DSLs without disrupting dev. chain



General Principles of Expression Templates

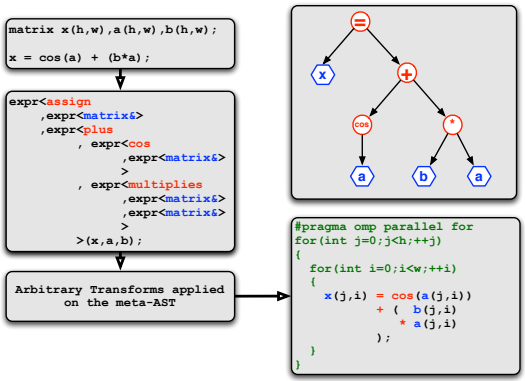
The Expression Templates Idiom

Principles

- Relies on extensive operator overloading
- Carries semantic information around code fragment
- Introduces DSLs without disrupting dev. chain

Advantages

- Generic implementation becomes self-aware of optimizations
- API abstraction level is arbitrary high
- Boilerplates reduced to acceptable level via tools



General Principles of Expression Templates

Tools of the Trade

Boost.Proto (pre-C++11)

- Designed by Eric Niebler [2008]
- Provides interface over the boilerplate of running Expression Templates
- Gave rise to Boost.Phoenix, NT2 and more

Boost.YAP (C++14 & onward)

- Designed by T. Zachary Laine [2018]
- Spiritual successor of Boost.Proto
- First Conceptualized Expression Templates helper

The early years

Blitz++, Veldhuizen et al.

- Probably the first recorded instance of ET in the wild
- Designed to make C++ as efficient as FORTRAN
- Implemented historically in C++98. Had a resurgence post C++11

The early years

Blitz++, Veldhuizen et al.

- Probably the first recorded instance of ET in the wild
- Designed to make C++ as efficient as FORTRAN
- Implemented historically in C++98. Had a resurgence post C++11

EIGEN, Guennebaud et al.

- First ET based algebra library to gain mainstream recognition
- Designed to support parallelism
- Slowly but surely migrate to C++11 and onward

The Next Generation

Blaze, Iglberger et al.

- Introduced Smart Expression Templates
- Design to solve both dense and sparse problems
- Heavily algebra biased

The Next Generation

Blaze, Iglberger et al.

- Introduced Smart Expression Templates
- Design to solve both dense and sparse problems
- Heavily algebra biased

NT2, Falcou et al.

- Uses Smart Expression Templates
- Designed to emulate MATLAB syntax
- Separate architectures from ASTs

The Next Generation

Blaze, Iglberger et al.

- Introduced Smart Expression Templates
- Design to solve both dense and sparse problems
- Heavily algebra biased

NT2, Falcou et al.

- Uses Smart Expression Templates
- Designed to emulate MATLAB syntax
- Separate architectures from ASTs

Armadillo, Sanderson et al.

- Uses dynamic optimisations strategies
- Designed to emulate MATLAB syntax

EDSL shortcomings

Interaction with auto

```
1 matrix A,B;
2
3 // What is the type of C ?
4 auto C = A * B;
```

Interaction with temporaries

```
1 template<typename T> auto f(T const& t)
2 {
3   matrix m = t + t;
4   return m / t; // Somebody set up us the bomb
5 }
```

Interaction with template types

```
1 template<typename T> auto g(T const& t)
2 {
3   T that = t * t; // Nuclear launch detected
4   return that;
5 }
```

EDSL shortcomings

DSEL: Well, we dont need to separate concerns

Users: You don't work in this simple situation

DSEL:

EDSL shortcomings

DSEL: Well, we dont need to separate concerns

Users: You don't work in this simple situation

DSEL:

Origins of those issues

- Most EDSL terminals act as both storage and tree spawner
- Most EDSL are opaque to type computation
- Most EDSL don't support sub-expression building in template contexts

EDSL shortcomings

DSEL: Well, we dont need to separate concerns

Users: You don't work in this simple situation

DSEL:

Our proposal

- Build a symbolic EDSL in which storage and expression are separated
- Make symbolic expression first class citizen within the library
- Support transformation of expressions

What is symbolic calculus?

Formulas

- Volume of a sphere:

$$V = \frac{4}{3}\pi R^3$$

What is symbolic calculus?

Formulas

- Volume of a sphere:

$$V = \frac{4}{3}\pi R^3$$

- Normal distribution PDF:

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

What is symbolic calculus?

Formulas

- Volume of a sphere:

$$V = \frac{4}{3}\pi R^3$$

- Normal distribution PDF:

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

- Riemann zeta function:

$$\zeta(s) = \frac{1}{\Gamma(s)} \int_0^\infty \frac{x^{s-1}}{e^x - 1} dx$$

What is symbolic calculus?

Formulas

- Volume of a sphere:

$$V = \frac{4}{3}\pi R^3$$

- Normal distribution PDF:

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

- Riemann zeta function:

$$\zeta(s) = \frac{1}{\Gamma(s)} \int_0^\infty \frac{x^{s-1}}{e^x - 1} dx$$

- Incompressible Navier-Stokes equation:

$$\frac{\partial \vec{u}}{\partial t} + \left(\vec{u} \cdot \vec{\nabla}\right) \vec{u} - \nu \vec{\nabla}^2 \vec{u} = -\vec{\nabla} w + \vec{g}$$

What is symbolic calculus?

Formulas

- Volume of a sphere:

$$V = \frac{4}{3}\pi R^3$$

- Normal distribution PDF:

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

- Riemann zeta function:

$$\zeta(s) = \frac{1}{\Gamma(s)} \int_0^\infty \frac{x^{s-1}}{e^x - 1} dx$$

- Incompressible Navier-Stokes equation:

$$\frac{\partial \vec{u}}{\partial t} + \left(\vec{u} \cdot \vec{\nabla}\right) \vec{u} - \nu \vec{\nabla}^2 \vec{u} = -\vec{\nabla} w + \vec{g}$$

- Einstein field equation:

$$R_{\mu\nu} - \frac{1}{2} R g_{\mu\nu} + \Lambda g_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}$$

Symbolic computing

Substitution

- Formulas are made of symbols, some of which are variables
- Substitution consistently replaces symbol variables by values or expressions

Symbolic computing

Substitution

- Formulas are made of symbols, some of which are variables
- Substitution consistently replaces symbol variables by values or expressions

Computer Algebra Systems (CAS)

Softwares manipulating analytical expressions (Mathematica, Maple, Sage, . . .)

Symbolic computing

Substitution

- Formulas are made of symbols, some of which are variables
- Substitution consistently replaces symbol variables by values or expressions

Computer Algebra Systems (CAS)

Softwares manipulating analytical expressions (Mathematica, Maple, Sage, . . .)

Examples

> `derivative`$(\sin(x) + x^2, x)$

Symbolic computing

Substitution

- Formulas are made of symbols, some of which are variables
- Substitution consistently replaces symbol variables by values or expressions

Computer Algebra Systems (CAS)

Softwares manipulating analytical expressions (Mathematica, Maple, Sage, ...)

Examples

> `derivative`$(\sin(x) + x^2, x)$
$2x + \cos(x)$

Symbolic computing

Substitution

- Formulas are made of symbols, some of which are variables
- Substitution consistently replaces symbol variables by values or expressions

Computer Algebra Systems (CAS)

Softwares manipulating analytical expressions (Mathematica, Maple, Sage, ...)

Examples

> `derivative`$(\sin(x) + x^2, x)$
$2x + \cos(x)$
> `simplify`$(\sin(x)^2 + \cos(x)^2)$

Symbolic computing

Substitution

- Formulas are made of symbols, some of which are variables
- Substitution consistently replaces symbol variables by values or expressions

Computer Algebra Systems (CAS)

Softwares manipulating analytical expressions (Mathematica, Maple, Sage, . . .)

Examples

> `derivative`$(\sin(x) + x^2, x)$
$2x + \cos(x)$
> `simplify`$(\sin(x)^2 + \cos(x)^2)$
1

Symbolic computing

Substitution

- Formulas are made of symbols, some of which are variables
- Substitution consistently replaces symbol variables by values or expressions

Computer Algebra Systems (CAS)

Softwares manipulating analytical expressions (Mathematica, Maple, Sage, ...)

Examples

> `derivative`$(\sin(x) + x^2, x)$
$2x + \cos(x)$
> `simplify`$(\sin(x)^2 + \cos(x)^2)$
1

It would be nice if only...

...we could do symbolic calculus inside a general purpose language like C++

Mathematical expressions

Formula

A mathematical expression assigned to a symbol. In other words, a mathematical expression that has been given a name.

$$V = \frac{4}{3}\pi R^3$$

So we need

- Support for constants, named constants, operators, and functions
- Support for formula combination

Mathematical expressions

Formula

A mathematical expression assigned to a symbol. In other words, a mathematical expression that has been given a name.

$$V = \frac{4}{3}\pi R^3$$

Equation

A mathematical statement that asserts equality between two expressions, for some values of the variable symbols.

$$ax^2 + bx + c = 0$$

So we need

- Support for constants, named constants, operators, and functions
- Support for formula combination

So we need

- Support for statement reorganization
- Support for variable or subexpression substitution

Mathematical expressions

Formula

A mathematical expression assigned to a symbol. In other words, a mathematical expression that has been given a name.

$$V = \frac{4}{3}\pi R^3$$

So we need

- Support for constants, named constants, operators, and functions
- Support for formula combination

Equation

A mathematical statement that asserts equality between two expressions, for some values of the variable symbols.

$$ax^2 + bx + c = 0$$

So we need

- Support for statement reorganization
- Support for variable or subexpression substitution

Identity

An equation that is always true.

$$\sin(x)^2 + \cos(x)^2 = 1$$

So we need

Support for expression/subexpression structural matching

Types of mathematical expressions

V • T • E	Arithmetic expressions	Polynomial expressions	Algebraic expressions	Closed-form expressions	Analytic expressions	Mathematical expressions
Constant	Yes	Yes	Yes	Yes	Yes	Yes
Elementary arithmetic operation	Yes	Addition, subtraction, and multiplication only	Yes	Yes	Yes	Yes
Finite sum	Yes	Yes	Yes	Yes	Yes	Yes
Finite product	Yes	Yes	Yes	Yes	Yes	Yes
Finite continued fraction	Yes	No	Yes	Yes	Yes	Yes
Variable	No	Yes	Yes	Yes	Yes	Yes
Integer exponent	No	Yes	Yes	Yes	Yes	Yes
Integer nth root	No	No	Yes	Yes	Yes	Yes
Rational exponent	No	No	Yes	Yes	Yes	Yes
Integer factorial	No	No	Yes	Yes	Yes	Yes
Irrational exponent	No	No	No	Yes	Yes	Yes
Logarithm	No	No	No	Yes	Yes	Yes
Trigonometric function	No	No	No	Yes	Yes	Yes
Inverse trigonometric function	No	No	No	Yes	Yes	Yes
Hyperbolic function	No	No	No	Yes	Yes	Yes
Inverse hyperbolic function	No	No	No	Yes	Yes	Yes
Non-algebraic root of a polynomial	No	No	No	No	Yes	Yes
Gamma function and factorial of a non-integer	No	No	No	No	Yes	Yes
Bessel function	No	No	No	No	Yes	Yes
Special function	No	No	No	No	Yes	Yes
Infinite sum (series) (including power series)	No	No	No	No	Convergent only	Yes
Infinite product	No	No	No	No	Convergent only	Yes
Infinite continued fraction	No	No	No	No	Convergent only	Yes
Limit	No	No	No	No	No	Yes
Derivative	No	No	No	No	No	Yes
Integral	No	No	No	No	No	Yes

Mathematical Abstract Syntax Trees (ASTs)

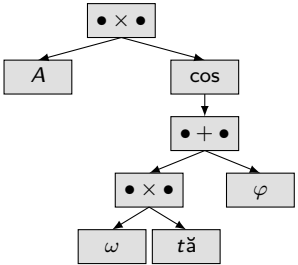
Formula example

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

Mathematical Abstract Syntax Trees (ASTs)

Formula example

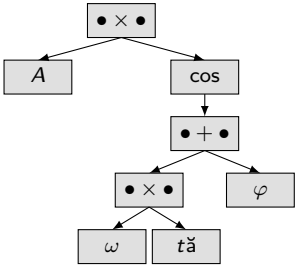
$$f = A\cos(\omega t + \varphi)$$



Mathematical Abstract Syntax Trees (ASTs)

Formula example

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



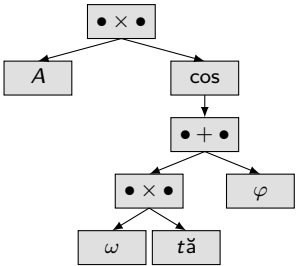
Tree vocabulary

- **level**: shortest path length from a node to root
- **arity**: number of children of a node
- **terminal**: a node without children

Mathematical Abstract Syntax Trees (ASTs)

Formula example

$$f = A\cos(\omega t + \varphi)$$



Tree vocabulary

- **level**: shortest path length from a node to root
- **arity**: number of children of a node
- **terminal**: a node without children

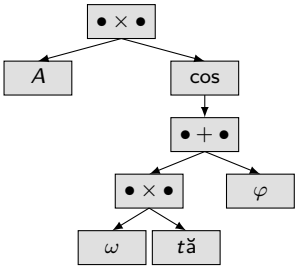
Formula components

- **expression**: a finite well-formed combination of symbols
- **subexpression**: a well-formed part of an expression
- **symbol**: an atomic element of an expression
- **functions/operators**

Mathematical Abstract Syntax Trees (ASTs)

Formula example

$$f = A\cos(\omega t + \varphi)$$



Tree vocabulary

- **level**: shortest path length from a node to root
- **arity**: number of children of a node
- **terminal**: a node without children

Formula components

- **expression**: a finite well-formed combination of symbols
- **subexpression**: a well-formed part of an expression
- **symbol**: an atomic element of an expression
- **functions/operators**

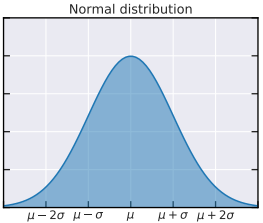
Types of terminal

- **nullary functions**: such as a random number generator
- **symbolic constants**: such as π
- **symbolic variables**: such as x, y, or z

Example of expression representation

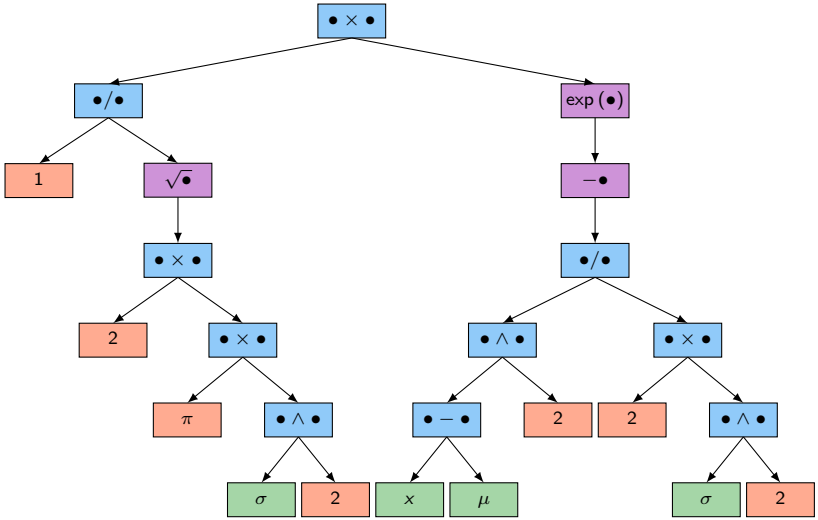
Normal distribution PDF

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



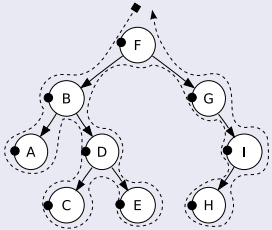
Legend

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



Visiting Abstract Syntax Trees

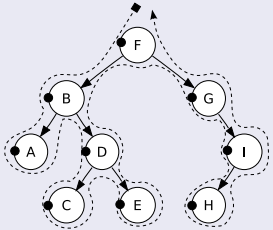
Depth-first pre-order



use: execution

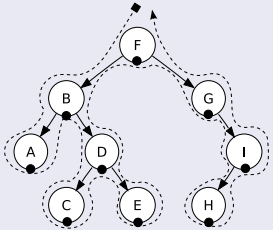
Visiting Abstract Syntax Trees

Depth-first pre-order



use: execution

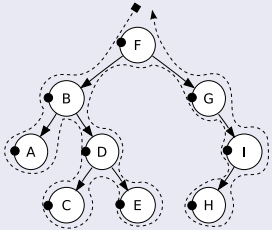
Depth-first in-order



use: display

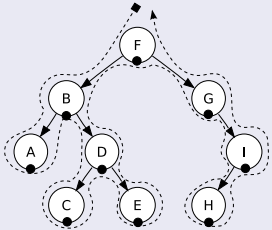
Visiting Abstract Syntax Trees

Depth-first pre-order



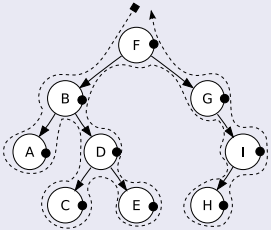
use: execution

Depth-first in-order



use: display

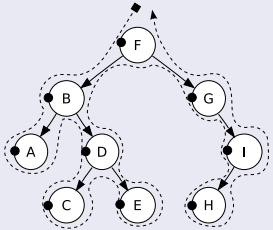
Depth-first post-order



use: expression analysis

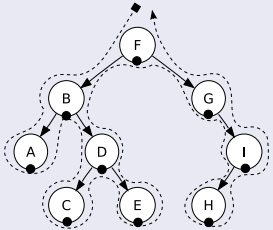
Visiting Abstract Syntax Trees

Depth-first pre-order



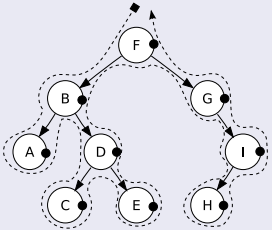
use: execution

Depth-first in-order



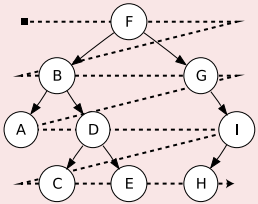
use: display

Depth-first post-order



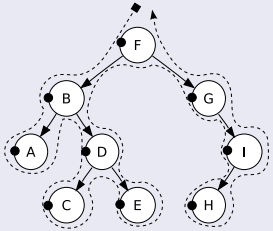
use: expression analysis

Breadth-first



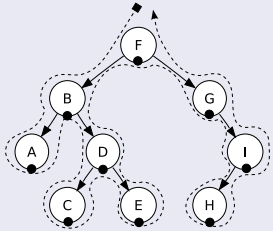
Visiting Abstract Syntax Trees

Depth-first pre-order



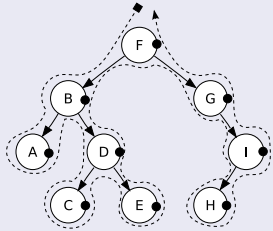
use: execution

Depth-first in-order



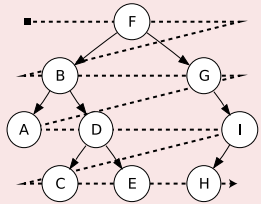
use: display

Depth-first post-order



use: expression analysis

Breadth-first



Generic tree traversal strategy

```
1  template <class Node, class Pre, class In, class Post>
2  void traverse(Node&& node, Pre prefunction, In infunction, Post postfunction)
3  {
4      prefunction(std::forward<Node>(node));
5      for (auto&& child: std::forward<Node>(node)) {
6          traverse(std::forward<decltype(child)>(child), prefunction, infunction, postfunction);
7          infunction(std::forward<Node>(node));
8      }
9      postfunction(std::forward<Node>(node));
10 }
```

Example of mathematical AST traversal

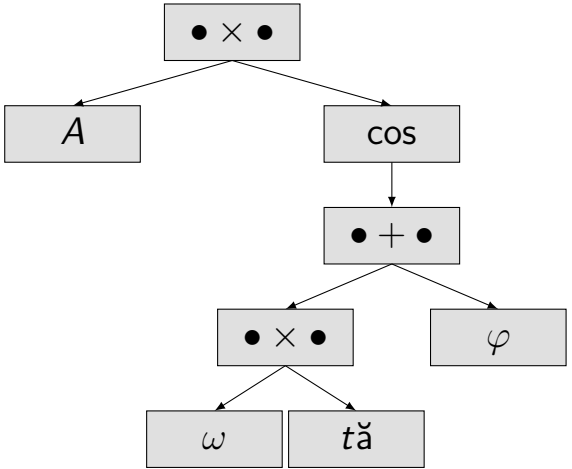
Mathematical expression

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

Example of mathematical AST traversal

Mathematical expression

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



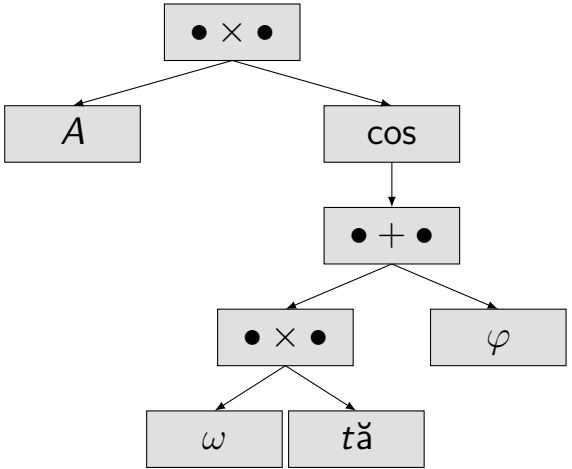
Example of mathematical AST traversal

Mathematical expression

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

Pre-order

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



Example of mathematical AST traversal

Mathematical expression

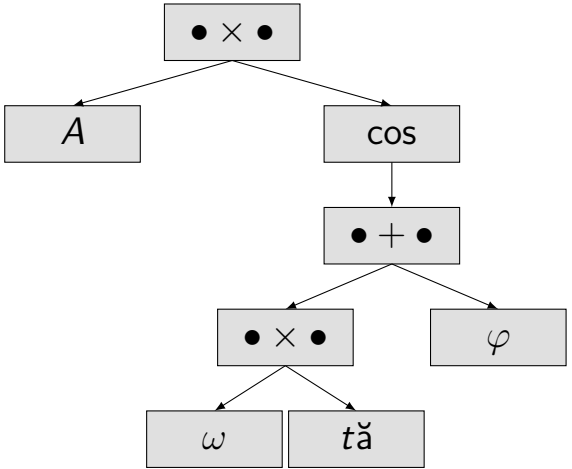
$$A\cos\left(\omega t + \varphi\right)$$

Pre-order

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

In-order

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



Example of mathematical AST traversal

Mathematical expression

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

Pre-order

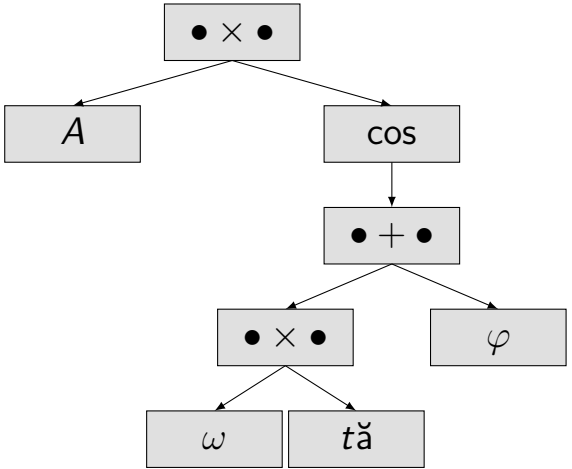
$$\times\left(A, \cos\left(+\left(\times\left(\omega, t\right), \varphi\right)\right)\right)$$

In-order

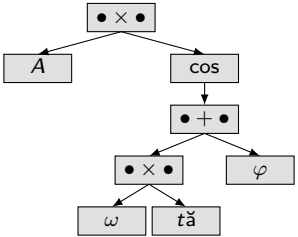
$$A\cos\left(\omega t + \varphi\right)$$

Post-order

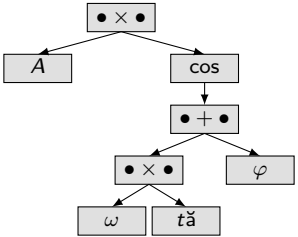
$$A, \omega, t, \times, \varphi, +, \cos, \times$$



Structure and data



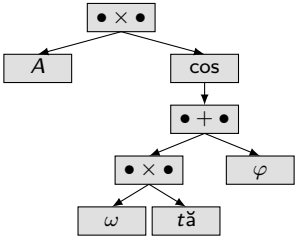
Structure and data



State

Mathematical expressions are **stateless**

Structure and data



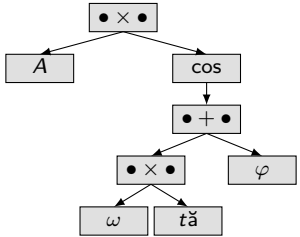
State

Mathematical expressions are **stateless**

Separation of concerns

Do not use data as terminals

Structure and data



State

Mathematical expressions are **stateless**

Separation of concerns

Do not use data as terminals

Solution

- Step 1: build the AST structure of the formula
- Step 2: inject data in the formula through symbol substitution

What do we want to achieve ?

Expected API support

- Build stateless formula from pre-existing or custom variables
- Interact with formula structure
- Visit formula ASTs to provide runtime behavior

Implementation requirements

- Expression templates with `constexpr` behavior
- Arguments passed by names
- Extensible visitor infrastructure
- Be as Simple Stupid C++17 as possible

What do we want to achieve ?

Expected API - Formula and Variables

```
1  // Basic formula
2  auto f = 6.67408e-11 * x_*y_ / ( z_ * z_ );
3  auto g = 6.67408e-11 * $(mass1)*$(mass2) / ($(distance)*$(distance));
4
5  // Variable renaming - g, g2 and f are equivalent
6  auto g2 = f( x_ = $(mass1), y_ = $(mass2), z_ = $(distance));
7
8  // Partial evaluation - g_10 is still a formula
9  auto g_10 = g( $(distance) = 10. );
10
11 // Mix different formulas together - k requires 6 variables
12 auto k = g + f;
13
14 // Full evaluations
15 auto qx = g( $(mass1) = 3.5, $(distance) = 10., $(mass2) = 7.6);
16 auto rx = g_10( $(mass1) = 3.5, $(mass2) = 7.6);
```

Lightweight Expression Templates

Implementation Strategy

- Reuse Boost.Proto model of tree, terminal, nodes + CRTP
- Reuse C++14/C++17 standard elements (traits, tuple)
- Stateless formula = Simpler expression implementation + `constexpr` everywhere possible
- Enforce compile-time with *ad hoc* solutions

Expression Templates - Node types

```
1  template<typename Tag, typename... Children>
2  struct node : expr<node<Tag, Children...>>
3  {
4    constexpr node(Children const&... cs) : children_(cs...) {}
5
6    static constexpr int arity()     noexcept { return sizeof...(Children);  }
7    static constexpr Tag tag()       noexcept { return {}; }
8
9    constexpr std::tuple<Children...> const& children()  noexcept { return children_; }
10
11   template<std::size_t Index>
12   constexpr auto get() const noexcept { return std::get<Index>(children_); }
13
14   private:
15   std::tuple<Children...> children_;
16 };
```

Lightweight Expression Templates

Expression Templates - Node types

```
1  template<typename Tag, typename Child> struct node<Tag, Child> : expr<node<Tag, Child>>
2  {
3    static constexpr int arity() noexcept { return 1;  }
4    static constexpr Tag  tag()  noexcept { return {}; }
5    template<std::size_t Index> constexpr Child get() const noexcept
6    {
7      return child0_;
8    }
9
10   Child child0_;
11 };
12
13 template<typename Tag, typename Child0, typename Child1>
14 struct node<Tag, Child0, Child1> : expr<node<Tag, Child0, Child1>>
15 {
16   static constexpr int  arity() noexcept { return 2;  }
17   static constexpr Tag  tag()   noexcept { return {}; }
18   template<std::size_t Index> constexpr auto get() const noexcept
19   {
20     if constexpr( Index == 0) return child0_;
21     if constexpr( Index == 1) return child1_;
22   }
23
24   Child0 child0_;
25   Child1 child1_;
26 };
```

AST Matching

Objectives

- Simplify inspection of ASTs
- Allow us to not require Boost.Proto like machinery for building grammar
- Must supports common placeholders liek any terminal, any operator, etc ...

Implementation

- Each expression type provide a match constexpr function
- Use local knowledge of structure to perform matching
- Helper macro ensure calls is always constexpr

AST Matching

Node match function

```
1  template<typename Tag, typename... Children>
2    struct node : expr<node<Tag, Children...>>
3    {
4      // ...
5
6      static constexpr bool match(type_t<any_expr>) noexcept
7      {
8        return true;
9      }
10
11     template<typename Other> static constexpr bool match(type_t<Other>) noexcept
12     {
13       return std::is_same_v<Other,any_node<arity()>>;
14     }
15
16     template<typename OTag, typename... OChildren>
17     static constexpr bool match(type_t<node<OTag, OChildren...>>) noexcept
18     {
19       return       std::is_same_v<Tag,OTag>
20                &&  (sizeof...(Children) == sizeof...(OChildren))
21                &&  ( (Children::match( type_t<OChildren>{} ) && ...)  );
22     }
23
24     // ...
25   };
```

Online Expression simplification

Motivation

- Compile time of AST ET is tie to AST depth
- Simplified AST leads to faster evaluations
- Automatic application of basic identities
- Similar to the clang InstCombine pass

Supported simplification

```
1 // Simplified as 2 * x_
2 auto f = x_ + x_;
3
4 // Simplified as 0
5 auto g = x_ * 0;
6
7 // Simplified as x_^4
8 auto h = x_ * x_ * x_ * x_;
```

Online Expression simplification

Simplification implementation

```
1  template<typename XLHS, typename XRHS>
2  constexpr auto build( tags::plus_ const&, expr<XLHS> const& lhs, expr<XRHS> const& rhs ) noexcept
3  {
4    using namespace literal;
5
6    if      constexpr( NUCOG_MATCH(lhs,rhs) ) return 2_c * lhs;
7    else if constexpr( NUCOG_MATCH(lhs,0_c) ) return rhs;
8    else if constexpr( NUCOG_MATCH(rhs,0_c) ) return lhs;
9    else if constexpr( NUCOG_MATCH(lhs,lit_ * expr_) && NUCOG_MATCH(rhs,lit_ * expr_))
10   {
11     if constexpr( NUCOG_MATCH(lhs[1_c],rhs[1_c]) )
12     {
13       using vl = decltype(lhs[0_c].value());
14       using vr = decltype(rhs[0_c].value());
15
16       return idx_<vl::value+vr::value>{} * lhs[1_c];
17     }
18     else
19     {
20       return node<tags::plus_,XLHS, XRHS>{lhs.self(), rhs.self()};
21     }
22   }
23   else
24   {
25     return node<tags::plus_,XLHS, XRHS>{lhs.self(), rhs.self()};
26   }
27 }
```

Symbolic variables

Implementation Strategy

- Symbolic variable only data is its name
- Custom variable requires arbitrary encoding of names
- Names are turned into types encoding the string via UDL

Symbolic variables API

```
1 // Manual definition
2 // Type is : symbol_id<integer_sequence<uint64_t, 13ul, 824199894539497688 4ul, 435610083689ul>>
3 auto var1 = "this_variable"_sym;
4
5 // Short-cut notation
6 // Type is : symbol_id<integer_sequence<uint64_t, 18ul, 730721841604233842 0ul,
7 //                                                  708906322844019109 0ul, 25964ul>>
8 auto var2 = $(the other variable);
```

Type-Value Maps

Problem

- Evaluation requires a name based argument passing
- We need to map a variable name to a value
- We need to retrieve those values later in visitor

Strategy

- Binding a type to a value is what functions do
- Generate a lambda for each variable substitution
- Aggregate lambda using a `overload`-like techniques
- Handles non-existing type substitution

Type-Value Maps

Type-Value aggregator

```cpp
template<typename T> struct box {};

template<typename... Ts>
struct aggregate_bindings : Ts...
{
  constexpr aggregate_bindings(Ts const&... t) noexcept : Ts(t)... {}
  constexpr aggregate_bindings(aggregate_bindings const& other) =default;

  using Ts::operator()...;

  struct not_supported_key {};

  template<typename K> constexpr not_supported_key operator()(box<K> const&) const noexcept
  {
    return {};
  }

  template<typename K, typename U>
  constexpr auto operator()(box<K> const& k, U const& u) const noexcept
  {
    // If calling without default would return the key, use the default
    if constexpr( std::is_same_v<decltype(this->operator()(k)),not_supported_key> )
      return u;
    else
      return this->operator()(k);
  }
};
```

Type-Value Maps

Type-Value wrapper via lambda

```
1  template<typename Key, typename Value> constexpr auto bind(Value&& v) noexcept
2  {
3    if constexpr( std::is_rvalue_reference_v<Value&&> )
4    {
5      return  [w = std::move(std::forward<Value>(v))](box<Key> const&) -> Value const&
6              {
7                return w;
8              };
9    }
10   else
11   {
12     return [&v](box<Key> const&) -> decltype(auto) { return v; };
13   }
14 }
```

Type-Value Maps

Typemap type

```
1  template<typename Storage> struct type_map_
2  {
3    template<typename K> constexpr decltype(auto) operator()(K const&) const
4    {
5      return storage(detail::box<K>{});
6    }
7
8    template<typename K, typename U>
9    constexpr decltype(auto) operator()(K const&, U const& u) const
10   {
11     return storage(detail::box<K>{}, u);
12   }
13
14   Storage storage;
15 };
16
17 template<typename... NamedParams> constexpr auto type_map(NamedParams&&... ts) noexcept
18 {
19   using s_t = decltype(detail::aggregate_bindings( std::forward<NamedParams>(ts)...));
20   return detail::type_map_<s_t>{detail::aggregate_bindings(std::forward<NamedParams>(ts)...)};
21 }
```

Type-Value Maps

Usage of type_map inside expression evaluation

```
1  // Call site
2  auto v = (x_ + y_)( x_ = .5f, y_ = 4);
3
4  // terminal association
5  template<typename T>
6  constexpr auto terminal<Symbol>::operator=(T&& v) const noexcept
7  {
8    return detail::bind<Symbol>(std::forward<T>(v));
9  }
10
11 // expression evaluation
12 template<typename... Params>
13 constexpr auto expr<Expr>::operator()(Params const&... ps) const
14 {
15   return evaluate( nucog::type_map(ps...), *this );
16 }
```

AST Visitor

Sample of a derivation visitor

```
1  template<typename V> struct derivate_visitor
2  {
3    template<typename X> constexpr auto visit(X const& expr) noexcept
4    {
5      if constexpr( NUCOG_MATCH(expr,term_) )
6      {
7        return nucog::as_expr(nucog::literal::idx_<NUCOG_MATCH(expr,var_)>{});
8      }
9      else if constexpr( NUCOG_MATCH(expr,expr_ + expr_ ) )
10     {
11       return visit(expr[0_c]) + visit(expr[1_c]);
12     }
13     else if constexpr( NUCOG_MATCH(expr, expr_ - expr_ ) )
14     {
15       return visit(expr[0_c]) - visit(expr[1_c]);
16     }
17     else if constexpr( NUCOG_MATCH(expr, expr_ * expr_ ) )
18     {
19       auto const& l = expr[0_c];
20       auto const& r = expr[1_c];
21       return visit(l)*r + l*visit(r);
22     }
23   }
24
25   private:
26   V var_;
27 };
```

Open Design Questions

Not sure if serious but...

- Is online simplification a good idea ?
- Processing array-like data requires support for broadcasting and global array operations
- Can we put semantic constraints on variables, e.g `auto f = tensor(x_) + real(z_);` ?

Possible solutions

- Provide a `simplify` visitor
- Implement a visitor that gather size informations
- Add thin wrapper that adds informative node in the AST

What did we learn?

Benefits of Symbolic EDSL

- Separation of concerns simplify EDSL implementation
- Stateless expression leads to simpler code
- Lambda as whatever is still going strong

Freebies we got on the trip

- Simple named arguments implementation
- EDSL expression constexpr matcher
- Simplified visitor boilerplate

What's next

Towards EDSL 3000

- Support Latex[1] as main EDSL
- Adds semantic informations on variables or formulas.
- Open doors for ODE and other solvers
- ... and don't get us started on the interactions with `std::embed`

See you at CppNow 2020 for the internals of those

```
1  // Build an expression by CT parsing of LaTex
2  auto f = "\sqrt[3]{\frac{x^3+y^3}{2}"_tex;
3  auto res = f(x_ = 4, y_ = covector({1,2,3,4}) );
4
5  // Solvers, Solvers everywhere...
6  auto m = solve( x_ + y_       = z_
7                , z_ + x_ - y_ = 0.5
8                , x_ - z_      = 0
9                );
```

[1] Thanks Hana Dusíková for the inspiration

Acknowledgments

This work has been made possible thanks to a number of people, organizations and institutions:

The Laboratory for Computation, Data, and Machine Learning at the University of Illinois (LCDM)

. And in particular Robert Brunner, Collin Gress, and Bryce Kille.

The National Science Foundation (NSF)

- Award NSF-SI2-SSE-1642411: Award Scalable Tree Algorithms for Machine Learning Applications
- Award NSF-CCF-1647432: EAGER: Next Generation Tree Algorithms

Institutions

- The *Laboratoire de Recherche en Informatique* at Paris-Sud University
- The Astronomy Department of the University of Illinois at Urbana-Champaign (UIUC)
- The Laboratory Universe and Theories (LUTH/OBSPM/PSL Research University)
- The National Center for Supercomputing Applications (NCSA)

Thank you for your attention