Oasis

User Manual

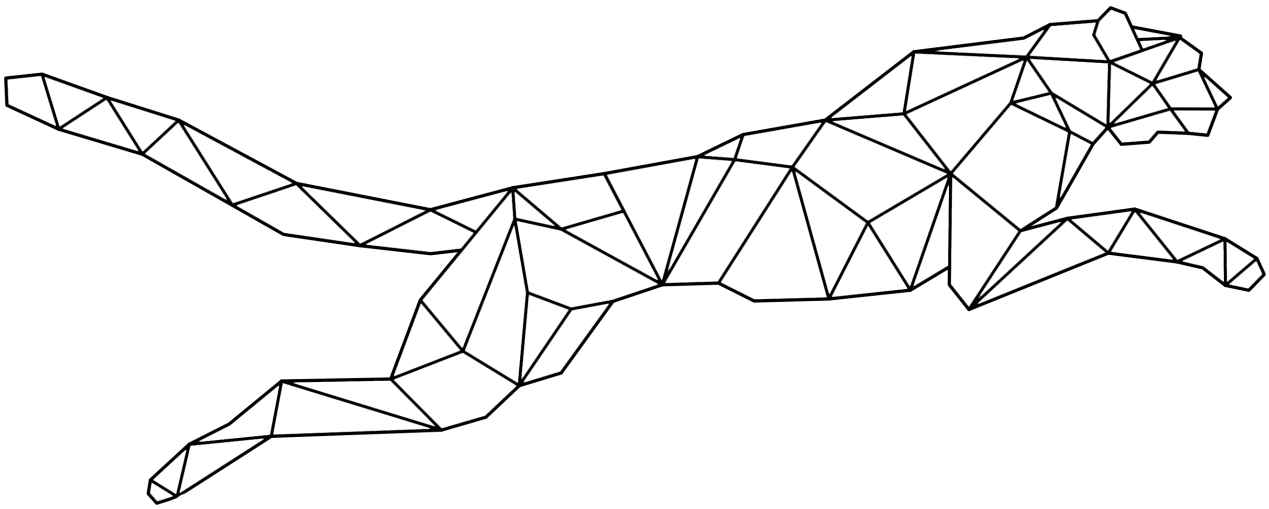

Table of Contents

Quick Startup

Step 1: Include the necessary Oasis Headers in your source file.

```
#include "Oasis/Add.hpp"
#include "Oasis/Exponent.hpp"
#include "Oasis/Imaginary.hpp"
#include "Oasis/Multiply.hpp"
#include "Oasis/Real.hpp"
#include "Oasis/Variable.hpp"
```

Step 2: Use the provided classes to create mathematical expressions.

```
Oasis::Add add {
    Oasis::Add {
        Oasis::Real {1.0},
        Oasis::Real {2.0}
    },
    Oasis::Real {3.0}
};
```

In this example, we define the addition of three numbers: 1.0, 2.0, and 3.0.

Step 3: Simplify the created expressions.

```
auto simplified = add.Simplify();
```

The Simplify() function simplifies the expression to its simplest form.

Step 4: Use the speciaized As<>() function to cast the simplified result to the expected type and retrieve the value.

```
auto simplifiedReal = simplified->As<Oasis::Real>();
auto value = simplifiedReal.GetValue();  // value will hold the result of 1.0 + 2.0 + 3.0
```

Step 5: Oasis also allows symbolic calculations. Here is how you can perform symbolic addition:

```
Oasis::Add add {
    Oasis::Multiply {
        Oasis::Real {1.0},
        Oasis::Variable {"x"}
    },
    Oasis::Multiply {
        Oasis::Real {2.0},
        Oasis::Variable {"x"}
    }
};

auto simplified = add.Simplify();
```

The code above performs the symbolic addition of “1x” and “2x”, simplifying it to “3*x”.

Step 6: You can also perform asynchronous simplification by using the SimplifyAsync() method.

```
std::unique_ptr<Oasis::Expression> simplified = add.SimplifyAsync();
```

Step 7: To check if two expressions are structurally equivalent, you can use the StructurallyEquivalent() or StructuallyEquivalentAsync() function.

```
bool areEquivalent = Oasis::Add {
    Oasis::Real {},
    Oasis::Real {}
}.StructurallyEquivalent(
    Oasis::Add {
        Oasis::Real {},
        Oasis::Real {}
    }
);
```

The above code checks if two addition expressions are structurally equivalent.

Important Concepts

Expressions: An expression in Oasis is a mathematical operation which could be as simple as adding two numbers or as complex as a symbolic computation.

Real and Imaginary Numbers: Oasis provides Real and Imaginary classes to represent real and imaginary numbers respectively.

Symbolic Computations: Oasis library supports symbolic computations allowing algebraic expressions to be represented and manipulated symbolically.

Asynchronous Computations: Oasis allows for asynchronous computation, which means you can perform operations in a non-blocking manner.

Structural Equivalence: Two expressions are said to be structurally equivalent if they have the same structure (same operations and ordering of operands). The StructurallyEquivalent() function checks this property.

By understanding these concepts and using the steps mentioned above, you should be able to comfortably use the Oasis library. Happy Coding!

Converting an Oasis::Expression Object to a MathML String in C++

This guide shows you how to convert an Oasis::Expression object into a serialized MathML string using C++ and the TinyXML2 library.

Step 1: Prepare XML Document
You'll need to create an instance of tinyxml2::XMLDocument. This object represents an XML file in memory and allows you to manipulate and extract information from it:

```
tinyxml2::XMLDocument doc;
```

Step 2: Create MathMLSerializer
Next, you'll need to create an instance of Oasis::MathMLSerializer. This class is used to serialize Oasis expressions into MathML:

```
Oasis::MathMLSerializer serializer { doc };
```

MathMLSerializer is initialized with the doc object which is a mold for the final XML code representation.

Step 3: Create XMLPrinter
After that, you'll need to create tinyxml2::XMLPrinter. This object is used to convert the document object model (i.e., the XMLDocument) to a string:

```
tinyxml2::XMLPrinter printer;
```

Step 4: Create Math Element and add to Document
Create a new element named “math” and set its attribute “display” to “block”. This element is the root element of a MathML document:

```
tinyxml2::XMLElement* mathElement = doc.NewElement("math");
mathElement->SetAttribute("display", "block");
doc.InsertFirstChild(mathElement);
```

After creating the root element, it’s inserted as the first child of doc.

Step 5: Serialize Oasis Expression
Use the Serializer object to serialize the Oasis::Expression referenced by expr:

```
expr->Serialize(serializer);
tinyxml2::XMLElement* queryElement = serializer.GetResult();
```

The Serialize() method modifies the state of serializer such that we can later retrieve the serialized form of expr by calling serializer.GetResult().

Step 6: Attach Serialized Element to Main Element
Insert the serialized Oasis expression as the first child of the “math” element:

```
mathElement->InsertFirstChild(queryElement);
```

Step 7: Generate MathML String
At this point, doc contains an XML representation of the Oasis expression. To extract this structure as a string, call doc.Print(&printer). After the print operation is complete, you can get the string by calling printer.CStr(). Here’s the final code snippet for this step:

```
doc.Print(&printer);
return printer.CStr();
```

This should return a string representing the Oasis::Expression object as MathML.

Instructions to Build the Oasis Project

Prerequisites:

1. Ensure you have a C++ compiler installed. Oasis requires at least CMake 3.18 to build, so make sure your compiler supports at least this version.
2. Git should be installed on your machine as some dependencies are fetched from their respective Git repositories.

Steps to Build Oasis:

1. Clone the Oasis project from its repository.
    a. Use the command:
    git clone https://github.com/open-algebra/Oasis
    b. Navigate to the cloned directory: cd Oasis

2. Setup a Build Directory
    a. Inside the Oasis directory, create a new directory named “build”: mkdir build

3. Configure the Project
    a. Run cmake -B build . This will configure the project and prepare to generate the build system, using the default generator and compiler on your environment.
    b. If you want to use a specific generator, specify it with -G. For example, to use the “Ninja” generator, run:
    cmake -G “Ninja” -B build .

4. Customize the Build options
    a. Oasis project provides several options that you can toggle ON/OFF according to your needs:
        1. OASIS_BUILD_EXTRAS: Enables building extra modules for Oasis.
        2. OASIS_BUILD_TESTS: Enables building unit tests for Oasis.
        3. OASIS_BUILD_WITH_COVERAGE: Enables building Oasis with code coverage enabled. Note that only the Clang compiler is currently supported for code coverage.

    These options are turned OFF by default.
    b. To turn these options ON, you can run CMake with -D. For example, to
    enable building of tests, run cmake -DOASIS_BUILD_ TESTS=ON -B build .

5. Build the Project
    a. Run cmake --build . This will start the actual build process. Ensure that you have the necessary permissions to read and write in your build directory.

6. Running Tests (If OASIS_BUILD_TESTS was set to ON)
    a. After successfully building the project, you can run the unit tests using ctest command.

Feel free to repeat steps 3-5 to re-configure and re-build the project using different options. Make sure to clear your build directory before doing so.

Tutorial for Using the C Wrapper for Oasis

Step 1: Creating Real Numbers

To create real numbers, the function Oa_CreateReal() is used passing an integer or float number as an argument.

```
struct Oa_Expression* real1 = Oa_CreateReal(2);
struct Oa_Expression* real2 = Oa_CreateReal(3);
```

Step 2: Performing Mathematical Operations & Simplifying

You can perform addition, subtraction, multiplication and division using Oa_CreateAddNF(), Oa_CreateSubtractNF(), Oa_CreateMultiplyNF() and Oa_CreateDivideNF() functions respectively. These functions take two number expressions (expressed as struct Oa_Expression*) as arguments and return the expression structure. Note these are the “non-freeing” functions. Which makes a copy of the operands you pass in. Please see the documentation for their “free-ing” equivalents

After performing the operations, use of Oa_Simplify(expr) function is done to simplify the expressions. It takes an expression structure and simplifies it.

```
struct Oa_Expression* add = Oa_CreateAddNF(real1, real2);
struct Oa_Expression* addResult = Oa_Simplify(add);
```

Step 3: Retrieving Result From Expression

The Oa_GetValueFromReal(expr) function is used to retrieve the resulting value from the expression. It takes your simplified result expression as an argument and returns the final required value.

```
assert(Oa_GetValueFromReal(addResult) == 5);
```

Step 4: Clearing Memory

To free up the memory that has been allocated for the different number expressions, use Oa_Free(expr).

```
Oa_Free(real1);
Oa_Free(real2);
Oa_Free(addResult);
```

This concludes the basic usage of Oasis library. Remember to properly handle memory allocation and deallocation to prevent any leaks.

Implementing an Operation

Welcome developer! So you want to contribute to Oasis? Great! If you're working on Simplify, Differentiate, or Integrate, you can think of these functions as performing pattern matching on the structure of trees.

The Specialize Function

For instance, suppose we have std::unique_ptr<Oasis::Expression> expr. We do not know much about expr, it could be anything. That's where the Specialize function comes in. We can give the Specialize function the type of the expression to check for, for example Oasis::Add<Oasis::Real> and it will return a std::unique_ptr to that if it satisfies the type or nullptr otherwise.

Example

Let's see it in action!

```
Oasis::Add add {
  Oasis::Real { 2 },
  Oasis::Real { 3 }
};

std::unique_ptr<Oasis::Expression> expr = add.Generalize();
// Oh no! we lost type information;

if (auto realCase = Add<Real>::Specialize(simplifiedAdd); realCase != nullptr) {
  const Real& firstReal = realCase->GetMostSigOp();
  const Real& secondReal = realCase->GetLeastSigOp();

  // We got type information back! Almost magical.
}
```

Now we do this a bunch of times for different scenarios, and boom! We are doing computer algebra.

Wildcards

Ok this works pretty well for $2+2$. Now what about $2x+2x$? You could check for Oasis::Add<Oasis::Multiply<Oasis::Real, Oasis::Variable and do what you need to do, but then what about $2(x+y)+2(x+y)$? You could spend eons writing each individual case, or you can use wildcards. Specialize treats Oasis::Expression as a wildcard. For instance:

```
if (auto likeTermsCase = Add<Multiply<Real, Expression>>::Specialize(simplifiedAdd); likeTermsCase != nullptr) {
  const Oasis::IExpression auto& leftTerm = likeTermsCase->GetMostSigOp().GetLeastSigOp();
  const Oasis::IExpression auto& rightTerm = likeTermsCase->GetLeastSigOp().GetLeastSigOp();

  if (leftTerm.Equals(rightTerm)) {
    const Real& coefficient1 = likeTermsCase->GetMostSigOp().GetMostSigOp();
    const Real& coefficient2 = likeTermsCase->GetLeastSigOp().GetMostSigOp();
    // Do what you need to do
    }
  }
```

Automatic Flipping

Now you may wonder about $2(x+y)+(y+x)2$. Do not worry, Specialize will try all possible orderings where appropriate for you! (Granted, it could be slow).

Expression as Binary Trees

How do you represent an equation in a computer? We could do with a string, but that no fun for anyone. Instead, we do it with trees. Here is how it goes, almost everything is a node in the tree: operations, numbers, variables, functions, and so on. The operands of an operation or arguments to a function are children of said operand or function in the tree.

The Naive Approach

Let us consider a simple tree implementation:

```
struct Expression
{
  Expression *left, *right;
};

struct Add: public Expression { };

struct Real: public Expression
{
  double value;
};
```

While the above works for simple cases, it quickly becomes very messy when we want do more complicated things. For instance, to get values of out the tree, we must first do a dynamic_cast to get a Real or an Add. This became tedious for more complicated trees.

The Templated Approach

What if we could simply “get” the numbers in an Add?
We could go with templates:

```
struct Expression { };

template <typename LeftT, typename RightT>
struct Add: public Expression {
  Left* left;
  Right* right;
};

struct Real: public Expression
{
  double value;
};
```

Great! Now can simply “get” the number on the left-hand side and we will a variable of type Real - assuming we have a number on the left-hand side. Passing this tree around also becomes tedious, what if instead of Add<Real, Real>, we had Add<Add<Real, Real>, Real>>? We could just pass an Expression* to it again, but then we’re back to the same problem.

The Oasis Approach

So it’s a lot easier to pass around Expression*, but using templates makes it easier to actually work with. What if we just did both? In Oasis, each Expression has a Generalize method that returns an std::unique<Oasis::Expression, the status quo of passing expressions around. When you want to operate on the expression, you can use Oasis::Specialize to check if an expression is of a certain type and cast it if so!