

PHM211s – Discrete Mathematics

Project Report Task 1 Group 28

Shams El-Din Mohamed Abdel-Monem – CSE 2101442

December 22, 2024

Abstract

This report demonstrates Task 1 program and the test case provided.

Contents

Front-End	3
Explanation	3
Examples	3
Documentation	5
Valid Argument	5
Expression Operations	5
Test Case	6
Program	6
The program result	7
Hand Calculated Result	7
Back-End	8
Brief	8
Classes	8
Simplified Algorithm (valid function)	9

Front-End

Explanation

Code offers a user-first easy experience to allow easy use of `valid` and `satisfiable` operations, simply declare variables as `aVariable`, use C++ binary operators on variables like `a | b` or `b & !c` and it automatically creates objects of `Expression` type that can then be passed to any other functions.

Examples

// Example 1

Variable p, q; *// Variables*

```
bool valid = Argument::valid(  
    // variables  
    {&p, &q},  
  
    // conclusion  
    p | q,  
  
    // premises  
    p  
);
```

```
std::cout << valid << std::endl; // outputs true
```

// Example 2

Variable p, q; *// Variables*

```

bool valid = Argument::valid(
    // variables
    {&p, &q},

    // conclusion
    p | q,

    // premises
    !p
);

std::cout << valid << std::endl; // outputs false

// Example 3
Variable p, q, r; // Variables

bool valid = Argument::valid(
    // variables
    {&p, &q, &r},

    // conclusion
    p >> r,

    // premises
    p >> (q | !r),
    q >> (p & r)
);

```

```
std::cout << valid << std::endl; // outputs false
```

Documentation

Valid Argument

Takes a vector of variables pointers, a conclusion expression and **any** amount of premises expressions.

```
template<typename... Rest>
bool Argument::valid(std::vector<Variable *> variables,
                    Expression &&conclusion, Rest &&... rest);

template<typename... Rest>
bool Argument::valid(std::vector<Variable *> variables,
                    Expression &conclusion, Rest &&... rest);
```

Expression Operations

Thanks to the use of C++ operator overloading, here is simplified list of operators and there results

Operator	Expression Result	Meaning
!a	Not(a)	Not the value of expression a
a & b	And(a, b)	The And-ing between a and b
a b	Or(a, b)	The Or-ing between a and b
a » b	IfThen(a, b)	If a then b; Implication
a <=> b	Iff(a, b)	If and only if a then b

Expression Operations Examples

```
!p & !q == And(Not(p), Not(q));
```

```
!(p & !(q <=> p)) == Not(And(p, Not(Iff(q, p))));
```

Test Case

Program

Variable

```
f("I played football"),  
s("I played basketball"),  
b("I ate breakfast"),  
h("I am happy");
```

```
bool valid = Argument::valid(  
    // variables  
    {&f, &s, &b, &h},  
  
    // conclusion  
    h | f,  
  
    // premises  
    f | b,  
    b | s,  
    h >> b  
);
```

```
std::cout << (valid ? "The Argument is valid." : "The argument is not valid.") << s
```

The program result

The argument is not valid.

Process finished with exit code 0

Hand Calculated Result

f	s	b	h	(f or b)	(b or s)	(if h then b)	(h or f)
0	0	0	0			1	
0	0	0	1				1
0	0	1	0	1	1	1	0 (invalid)
0	0	1	1	1	1	1	1
0	1	0	0		1	1	
0	1	0	1		1		1
0	1	1	0	1	1	1	0 (invalid)
0	1	1	1	1	1	1	1
1	0	0	0	1		1	1
1	0	0	1	1			1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	1
1	1	0	0	1	1	1	1
1	1	0	1	1	1		1
1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1

Since there are at least 1 row where all the premises are true, the conclusion is false, **then this is a not a valid argument.**

Back-End

Brief

Thanks to heavy use of C++ **Object-Oriented Programming**, **operator overloading**, **method overriding** and **r-value references**, all code functions are encapsulated and have an intuitive easy to use interface to allow for **very general use cases**.

Classes

Most important class is the **Expression abstract** class, encapsulates the operator overloads, **evaluate** method used to evaluate the binary value of any expression offers extra methods for truth set generation and intersection.

Variable class is an **Expression** that has an exact value, a variable should always be kept passing a reference so that changing it once would affect all expressions using it.

BinaryExpression abstract composite class is an **Expression** that has two operands, offers elegant constructors with **r-value references** to limit use of **new** and **pointer** variables in user front-end code.

UnaryExpression abstract composite class is an **Expression** that has one operand, offers elegant constructors with **r-value references** to limit use of **new** and **pointer** variables in user front-end code.

And, Or, IfThen, Iff are **concrete** classes that are **BinaryExpression**, they implement **evaluate** to reflect there operation

Not is a **concrete** class that is a **UnaryExpression**, it implements **evaluate** to reflect its operation

Simplified Algorithm (valid function)

1. get truth set of conclusion expression
2. get the truth set of first premise expression
3. loop on all other premise expressions and intersect into one truth set
4. intersect the premises truth set and the conclusion truth set
 1. If the number of 1 s in premises the same after the intersection, this means the conclusion has at least the same 1 s in its truth set (if not more).
 2. If not, then there is at least one critical row where the premises truth are 1 and the conclusion was 0 .