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

Explanation

Code offers a user-first easy experience to allow easy use of `valid` and `satisfiable` operations, simply declare variables as a`Variable`, 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
```

```cpp
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)
        );
```

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

//Example 4
Variable p, q,r;

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

            // expressions
            p | !q,
            !p | q,
            p | q,
            !p | !q | !r
      );

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

Documentation

- The code minimal C++ version is version C++11
- The code contains C++20 overloading that is commented out for compatibility with most online compilers

Valid Argument

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

```cpp
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);
```

Satisfiable Argument

Takes any amount of expression

```cpp
template<typename... Rest>
static bool satisfiable(std::vector<Variable *> variables,
                        Expression&& first, Rest&& ... rest);

template<typename... Rest>
static bool satisfiable(std::vector<Variable *> variables,
                        Expression& first, 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

Operator	Expression Result	Meaning
a <=> b (C++20 and above)	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 satisfiable = Argument::satisfiable(
            // variables
            {&f, &s, &b, &h},

            // expressions
            h | f,
            f | b,
            b | s,
```

```
            h >> b
      );

bool valid = Argument::valid(
            // variables
            {&f, &s, &b, &h},

            // conclusion
            h | f,

            // premises
            f | b,
            b | s,
            h >> b
      );

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

The program result

```
The argument is satisfiable.
The argument is falsifiable.

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 falsifiable argument**. Since there are at least 1 row where all the statements are true, **then this is a satisfiable 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

Truth set function algorithm

Given a vector of `Variables` it tests an expression for all possible values of input and generates the rows in the truth table that are true, **it is guaranteed to be generated in order**.

1. generate all possible permutations of `VariableValue`
2. loop on all possible values and test the expression
   1. If it is true add to the truth set
   2. If it is not, skip

Truth set intersection function algorithm

1. defines two indices (i, j) for each truth sets (a, b) starting at first row (i = 0. j = 0)
2. compare the i-th and j-th entries in first and second truth sets respectively
   1. if they are equal (a[i] == b[j]), add to the final intersected truth set, increment both i and j
   2. else if a[i] > b[j] then increment j only
   3. else (a[i] < b[j]) then increment i only
   4. repeat the whole comparison step until indices are out of range

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 (valid

argument).

2. If not, then there is at least one critical row where the premises truth are *1* and the conclusion was *0* (falsifiable argument).

Simplified algorithm (satisfiable function)

1. get truth sets of all expressions
2. loop on all expressions and intersect into one truth set
3. intersect the premises truth set and the conclusion truth set
    1. If the number of *1* s in the intersection truth set is more than 1, then there is at least 1 common row between all expression (satisfiable),
    2. If not then there is no common row between all statements (not satisfiable)