

Java Term Rewriting System

Richard Whitty

June 2008

Abstract

This report details the implementation of a term-rewriting system for boolean algebra using the java programming language. It is capable of parsing propositional logic formula to an abstract syntax tree and performing a number of manipulations on the resulting tree, or generating a truth table for the given formula.

Such manipulations can be performed in an interactive manner, whereby the user is presented a menu listing the applicable rewrites. Alternatively there is support for planner classes which, using a generic interface, work towards some end result - for example converting into conjunctive or disjunctive normal form.

Additionally I propose an algorithm (with implementation) for verification of propositional arguments.

1 Motivation

During June I began experimenting with syntax trees, intending to use the work to improve my BrainFuck compiler¹. To do this I needed to develop both methods to build a tree, and formulate patterns to manipulate it in a generic manner.

At the same time I was approached by a student from the University of St Andrews who was attempting to construct a framework to manipulate boolean expressions, but was struggling to build a workable tree or parser. We initially planned to collaborate, but ended this due to differences in the pace of development.

2 Implementation

As always, when implementing a piece of software, there are certain trade offs that must be made, I detail them here.

Java was chosen as the implementation language as I had already developed an infix expression parser which could be easily adapted to parse the input.

2.1 Tree Structure

The tree nodes all inherit from an abstract base class *Clause*, which represents a complete tree or sub tree (so as to avoid the need for a *TreeRoot* class or similar, and also to allow operations to be performed on a specific part of the tree).

The boolean connectives, such as the logical operators *And*, *Or* and *Not* extend a subclass of *Clause* called *Connective*. Each of these provide a static traits object, which is used to determine the properties (such as the arity, whether the connective is distributive, commutative etc).

Also extending the *Clause* class are the terminals *Symbol* (which represents an atom or variable) and *Certainty*, which can be substituted for a sub tree should it be determined to always evaluate to a single truth value irrespective of inputs.

Of these, only subclasses of *Connective* are allowed to have child nodes. Attempting to add a subnode to a *Symbol* or *Certainty* results in a run-time error. (In java this is done using a *RuntimeException* as it would likely be the result of programmer error rather than user error)

2.2 Actions

These are operations which can run on a clause. They may or may not change it, but are intended to be called on any node of any sub tree (generally all of them in turn). It provides the following virtual function:

```
abstract Clause apply(Clause c);
```

2.3 Operations

Operations represent primitive operations based on the rules of propositional logic. These all inherit off the abstract class *Operation*, which extends *Action* and provides an addition method to check if the operation is applicable for the called sub tree.

```
boolean canApply(Clause c)
```

This can be used to construct a list of all applicable operations on a tree, which in turn can be used to select the appropriate transformation to perform.

2.4 Planners

When there is a specific goal in mind, such as converting to normal form, there are generally a few rules that need to be applied repeatedly in a specific order. It makes sense to create a generic interface to these sets of rules, for which I have arbitrarily chose the name '*Planner*' (as it 'plans' the path to take).

Again, the planner inherits from *Action* as it is to run on every node in the tree and decide what to do with it.

Where a possible application of an *Operation* is found, the planner creates an *ActionPlan* object, which stores both the *Operation* and *Clause* in order to present a choice at the end.

After being run on the whole tree, this results in a list of all the operations which can be performed. The default implementation takes the first one and runs that.

Clearly, modifying part of the tree can invalidate other *ActionPlans*, so it is necessary to re-run the planner to get the next set. The process is deemed to be complete when there are no more transformations available.

In the current version, there are three Planners, which differ by the rules they employ:

AllPlanner This includes all the rules. It will not terminate if used as an automatic one, but is useful for the interactive rewriting mode.

DNFPlanner This converts the tree into disjunctive normal form (DNF), that is, push the *Ors* to the top of the tree, and *Ands* to the bottom, with *Nots* and *Symbols* below them.

CNFPlanner This converts the tree into conjunctive normal form (CNF), that is, push the *Ands* to the top of the tree, and *Ors* to the bottom, with *Nots* and *Symbols* below them.

2.5 Argument Verifier

Propositional arguments of the form $\{ p1, p2, \dots, px \mid \text{con} \}$ can be parsed and partially verified. Where px are propositional formulae as parsed in the rest of the code (the assumptions), and con is the conclusion.

The idea is to test that the conclusion can be derived from the assumptions. A symbol table is constructed to store the assumed truth value of each symbol using an associative array, taking into account the connectives involved.

The method `addToKB()` walks the tree recursively while maintaining a variable `truthstate`, which is the truth value of any symbols encountered at that time. It negates `truthstate` and descends if a *Not* is encountered. If an *And* is encountered, it will descend if `truthstate` is true. If an *Or* is encountered, it will descend if `truthstate` is false.

```
values <- associative array of symbols to boolean values.
void addToKB(Clause c) { addToKB(c,true); }
void addToKB(Clause c,boolean truth) {
    if c is a NOT operator
        addToKB(c.child,! truth)
    else if truth is true and c is an AND operator

        addToKB(c.leftchild,truth)
        addToKB(c.rightchild,truth)
```

```

else if truth is false and c is an OR operator

    addToKB(c.leftchild,truth)
    addToKB(c.rightchild,truth)
else if c is a symbol
    values.set(c,truth)
}

```

After this has run for all the arguments (and no contradictions are discovered) the symbols in the conclusion are replaced with the truth values from the symbol table (using instances of Certainty) and folded to remove the branches. If the conclusion evaluates to FALSE, there is a contradiction. If it evaluates to TRUE, then the argument is deemed to be valid. Otherwise we determine that it is either ambiguous or beyond the capabilities of this program.

2.6 User interface

There are four modes of operation:

ConvertToCNF This reads in a formula, parses it and converts it to CNF using CNFPlanner.

ConvertToDNF converts to DNF using DNFPlanner

Actions Reads in a formula, then scans it using AllPlanner. The user is then presented with a menu asking which operation to perform, or to generate a truth table.

TruthTable Reads in a formula and produces a truth table.

Each of these will run until end-of-file (Ctrl+D).

3 Evaluation

3.1 What went right

- The program successfully parses the expression and builds the tree.
- It converts formulas into the normal forms based on the Planners and Operations.
- The argument verifier works correctly for the test data I have used. This includes the base cases of Modus Ponens, Modus Tollens, Modus tollendo ponens and Modus tollendo tollens. In the test cases the implication operator was input using the basic AND and NOT operators.

3.2 What went wrong

The use of inheritance for a tree is a double edged sword. While it affords polymorphism, there are also cases (for example when actually performing the rewrites) that you need to know the type of a certain node. In Java, this means using the `instanceof` operator, which is normally considered bad practice (as it defeats the purpose of inheritance - each should be treated the same).

This choice came about as in the BrainFuck compiler, I used tagged classes which produced some extensibility issues (addition of a node type meant multiple changes spread throughout the source base) and (in C++ at least) down-casting (to a derived type, e.g. with `dynamic_cast<>`) is considered an anti-pattern.

3.3 Possibilities for future work

As can be seen with the implementation of the implication operator, most can be expressed as a combination of others. Along with DeMorgan's laws, this could be used to reduce the set of necessary operators. For example it could be re-implemented so the only operators present in the tree structure are AND and NOT.