Imperial College London

Final year project

Equivalences Teaching Tool

Anna Thomas (at711)

CID: 00687463

June 15, 2014

Abstract

One logical formula can be converted into another using logical equivalences. In this report, we describe a teaching tool to help students new to logic understand propositional and first order logical equivalences.

We created a parser for propositional and first order logic to generate a tree which can be manipulated to apply equivalence rules. Using this underlying structure we created an Android app to convert one formula to another. The formation tree of the current formulae are shown and help is provided to encourage practice and understanding of equivalences.

Acknowledgements

Contents

Introduction

1.1 Motivation

All logics, including first order logic, are based on propositional logic in some form, so it is important that new students learn how to use it. New students learning propositional logic can struggle to understand the rules and how they should be applied to formulae. Having been through this situation myself and after speaking to my peers I understand the difficulties that can be had learning the concepts.

To help with this our idea is to create an equivalence teaching tool. This will be a tablet application which will allow a user to apply rules to a formula until they have reached the desired equivalence.

1.2 Approach

We decided early on in the project that the tool should be an Android tablet application as opposed to a web or mobile app (Chapter 6). This allows for an intuitive, interactive design while still having a large screen space.

The tool can be divided up into its main component parts: the parser, tree builder, tree processing and tablet interface. The parser will be generated by the *ANTLR4* parser generator, the tree builder, processing code and the tablet interface will be written in Java with the interface using the Android SDK.

The parser requires a grammar to generate the relevant parser components. This will be used to parse the initial equivalence and return the *ANTLR4* tree representation of the string (Chapter 3).

The tree builder is required to take the *ANTLR4* representation of a tree and convert it into a more useful data structure which can be modified and displayed easily (Chapter 4). The operators and atoms of the formula will be represented as nodes and leaves respectively.

Tree processing will be used to internally calculate which rules are applicable to each node and will allow us to subsequently manipulate the tree by applying these rules

(Chapter 5).

The interface will have an functional design displaying the current formula's formation tree (generated by the tree builder) and allowing a user to click on a node to apply a rule (Chapter 8).

1.3 Contributions

In this report we present a logic equivalence teaching tool to be used on an Android tablet, which uses propositional logic to allow a user to prove that two formulae are equivalent. The solution employs a generated parser with a custom logic grammar to generate the internal tree representation of a formula as well as providing tree manipulation to apply equivalence rules to the resulting tree.

The application contains some key features. These are outlined below:

1. **Graphical tree representation of formulae**

   Representing the formula as its formation tree will allow a user to see exactly how the formula is constructed and can help them understand the order of precedence of operations. It also allows the user to click on an operator to select a rule for it. This is much easier than clicking on a section of the formula to apply a rule to.

2. **Undo/Redo functionality**

   Previously equivalent formulae will be displayed above the current formula. When an old formula is selected it will expand into tree form and the formulae below it are faded out (Undo). This will allow a user to perform rules on the old tree or select one of the later faded trees (Redo). When a rule is applied, the faded formulae will be removed from the history allowing the user to continue on from that point (Figure 1.1).

/home/anna/workspace/EquivalencesApp/report/i/undo.png

Figure 1.1: Undo functionality - Clicking a previous equivalence shows the formula tree for that equivalence and sets the undone history as translucent. Applying a rule then overwrites the undone history.

3. **Generated equivalences**

   We want to allow the user to have equivalences automatically generated for them to solve. This would be implemented by running the tool on a generated formula to give a significantly different equivalent one. The key advantage of having this feature as well as allowing manual entry of equivalences would be that the user will have a continuous supply of new equivalent logic formulae. This also requires generating the initial formula for the tool to be run upon or allowing the user to manually enter the first formula and have the second one generated for them.

4. **Difficulty setting**

   Extending the idea of generated equivalences, we could allow the user to select a difficulty. This would be calculated by the length of the generated formula and how many rules were applied to get its equivalent formula.

5. **Help**

   Providing the user with help is key to their improvement. Mistakes such as cycles in equivalences should be pointed out to the user. We will also provide help by only displaying the rules that can be applied to the selected node instead of the whole set.

Background

2.1   Propositional Logic

Propositional logic is a branch of logic that studies ways of combining and modifying whole sentences, statements or propositions to form more complex propositions. It is a formal system containing logical relations and properties which are derived from these methods of joining or altering statements.

A logical system contains three major parts:

1. **Syntax** - the formal language that is used to express concepts.

2. **Semantics** - provide meaning for the language.

3. **Proof theory** - provides a way to convert one formula into another using a defined set of rules.

Logical definitions:

- *Atomic* - A formula whose logical form is $\top$, $\bot$ or p for an atom p.

- *Negated atomic* - A formula of the form $\neg p$.

- *Negated formula* - A formula of the form $\neg A$ for a formula A.

- *Conjunction* - A formula of the form $A \wedge B$.

- *Disjunction* - A formula of the form $A \vee B$.

- *Implication* - A formula of the form $A \rightarrow B$, where A is the *antecedent* and B is the *consequent*.

- *Literal* - A formula that is either atomic or negated atomic.

- *Clause* - A disjunction of one or more literals.

A *statement* is defined as a meaningful declarative sentence that is either true or false. For example, a statement could be:

- 'Socrates is a man.'

- 'All men live on Earth.'

A statement can be constructed of multiple parts, for example, the above statements can be combined into:

- 'Socrates is a man and all men live on Earth.'

Each part of this statement can be considered a proposition. Propositional logic involves studying the connectives that join these such as *'and'* and *'or'* (to form conjunctions and disjunctions), the rules that determine the truth values of the propositions and what that means for the validity of the statement.

2.2 First Order Logic

First order logic is differentiated from propositional logic by its use of *quantified variables* and *predicates*.

- A *quantified variable* is a variable bound over a *domain of discourse* (the set of entities over which quantified variables can range), it is bound by an operator called a *quantifier*. The *quantifier* is a type of determiner indicating quantity. There are two types in predicate logic: the universal quantifier '∀' ('all') and the existential quantifier '∃' ('exists').

- A *predicate* uses an entity from the domain and returns true or false.

Adding to propositional logic, first order logic also includes the quantifier symbols '∀' and '∃' in its alphabet as well as predicate and function symbols such as 'P' and 'f'. Predicate symbols are usually represented by upper case letters P, Q, R, etc. Predicate and function symbols have a *valence* (number of arguments), for example P(x) has valence 1.

In a *well-formed formula* (wff), variables can either be *free* or *bound*. If a variable is quantified it is classed as bound, otherwise it is a free variable. A *sentence* is a formula with no free variables.

2.3 Truth tables

It is necessary to understand the meanings of the symbols used in a language. Truth tables are mathematical tables used in logic to compute the functional values of logical expressions. They can be used to determine whether or not a propositional logic statement is logically valid.

A *situation* determines whether each propositional atom is true or false. A truth table shows all the situations the input variables can be in. We write 1 for true and 0 for false as shown below in Table 2.1:

9

A	B	A∧B	A∨B
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

Table 2.1: Truth table for A∧B and A∨B

Truth tables can be used to define any operator in propositional logic, including any new ones which might be desired. However, truth tables cannot be used with first order logic. With quantifiers, there is not finite set of propositions and variables do not refer to specific items in the domain.

2.4 Related Work

2.4.1 Logic Daemon

Created in *Texas A&M University*, the *Logic Daemon*[8] is an online logic proof checker. It comprises a simple web page with two small text input boxes for the premises and conclusion and then one large text box for applying primitive rules (Figure: 2.1).

While this tool does allow us to apply rules to prove equivalences we do not find it very intuitive to use at all. The interface itself looks quite unclear and isn't user friendly so it would not be suited to new students. Applying the rules to the premises is also quite confusing as it shows all the rules and does not alert the user to the fact they cannot be applied until they have already clicked on it. We found this a frustrating way of attempting a proof; consequently, we intend to only display the rules that can be applied in the current situation.

However, it is a useful tool for those more knowledgeable about logic for checking proofs once they have got to grips with the interface. It also has a *"Get Help"* button to suggest which rule to use next. We have used the idea of a help button in our project but to build upon it in order to provide more extensive help such as showing when an action has been repeated (a cycle has been reached).

We think that our tool feels more intuitive to use because we display the formulae as formation trees with interactive nodes to apply rules with. This provides a more straight forward way of presenting the rules to new students.

The website also provides a simple equivalence checker, well-formed formula checker and countermodel checker. These are all similar tools useful in logic but again the interface is not particularly intuitive or user friendly.

The *Logic Daemon* tool also provides support for first order logic which we have implemented as an extension.

/home/anna/workspace/EquivalencesApp/report/i/logicdaemon.png

Figure 2.1: Logic Daemon

2.4.2 Pandora

Pandora[9] is a tool created by Imperial College London; it stands for Proof Assistant for Natural Deduction using Organised Rectangular Areas. It can be used to prove that a goal formula follows from the given formulae. Pandora allows the user to repeatedly apply natural deduction rules (Figure 2.2).

We have not included natural deduction in our tool as we wanted to focus on solving propositional logic equivalences. *Pandora* is a much nicer tool than the *Logic Daemon* and we find it much more intuitive and easy to use.

Pandora allows a user to apply rules forwards or backwards. We know from using *Pandora* that some proofs are easier to complete working backwards. This is an idea we have incorporated into our tool by allowing the user to expand both the top and bottom formula to show their formation trees to apply rules.

/home/anna/workspace/EquivalencesApp/report/i/pandora.jpg

Figure 2.2: Pandora

2.4.3 Logic Solver

This is an Android application to show truth tables and possible logical equivalences. It looks like it has been made as a phone application as on a tablet it fills very little of the screen and it is difficult to click on some of the links (Figure 2.3).

/home/anna/workspace/EquivalencesApp/report/i/logicsolver.png

Figure 2.3: Logic Solver Android Application

The user can apply rules to the formula they enter into the application; this is done by selecting the rule they want out of a list which is then applied to the formula. However, they cannot use this application to solve an equivalence as they can only enter one formula and then apply rules to that.

The list of rules that is offered to the user only shows the rules that can be applied to the current formula. We have also offered this behaviour as there are far too many rules to display them all to the user and expect them to sort through which could be applied.

Formulae are displayed simply in the application. This could be improved upon so we are displaying the current formulae with their formation trees.

2.4.4 Truth Tables

Truth Tables is also an Android application mainly for use on a phone. It simply generates and displays truth tables based on a logical formula (Figure 2.4).

/home/anna/workspace/EquivalencesApp/report/i/truthtables.png

Figure 2.4: Truth Tables Android Application

We have not added the ability to view truth tables to our application, although we have implemented them to check formulae are equivalent. However, this could always be added as an extension.

2.4.5 LogicCalc

LogicCalc is an Android application for solving problems with propositional logic. It allows the user to create workbooks to save and print their proofs (Figure 2.5). However, we had trouble running it on our Nexus tablet.

/home/anna/workspace/EquivalencesApp/report/i/logiccalcapp.png
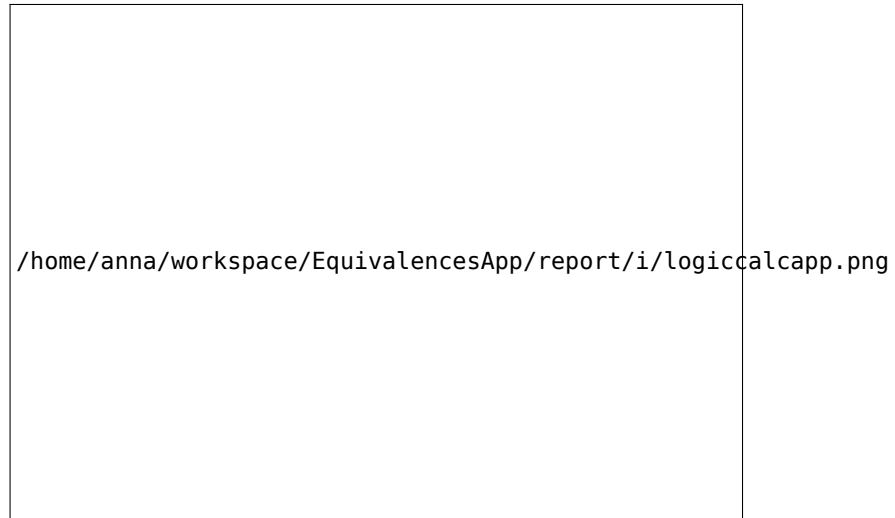
Figure 2.5: LogicCalc Android Application

We liked the idea of saving proofs in workbooks for future reference. This is something we considered implementing, but decided to go in other directions. This functionality would be useful for students completing exercises that needed a paper hand in as they would be able to save and print them off.

2.4.6 Propositional Logic Calculator

The *Propositional Logic Calculator*[10] finds all of the models of a given propositional formula. The website tells us that the only limitation for this calculator is that we have only three atomic propositions to choose from: *p*, *q* and *r* (Figure: 2.6).

Propositions are entered using the keyboard they provide on the calculator and the reasoning process is initiated by clicking 'ENTER'. It then calculates the truth value assignments that will make the formula true in the 'MODELS' section and the truth value assignments making the formula false in the 'COUNTERMODELS' section.

/home/anna/workspace/EquivalencesApp/report/i/logiccalc.png
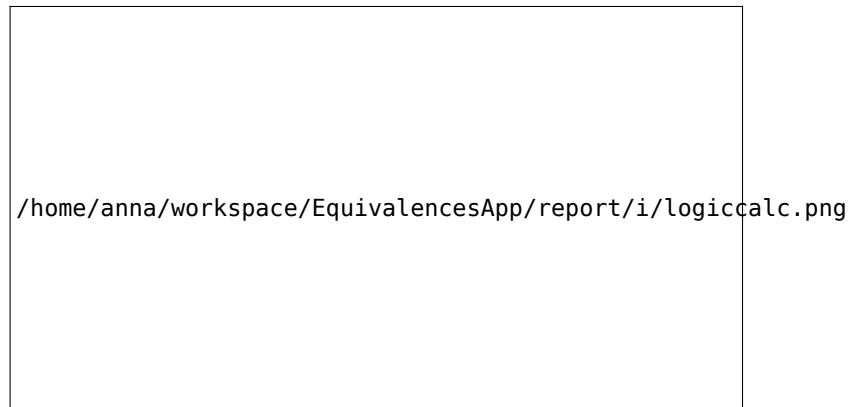
Figure 2.6: Propositional Logic Calculator

2.5 Parsers

Parsing is the process of breaking down data into smaller elements according the rules of a formal grammar. A parser will normally build a data structure such as a parse tree or abstract syntax tree from the elements. It is usually used as a component of a compiler and comprises of three parts:

1. Lexical analysis: A lexer creates tokens from the input stream.

2. Syntactic analysis: Uses a context-free grammar to check that the generated tokens form a meaningful expression.

3. Semantic parsing: Takes the validated expression and performs the necessary actions based on their meaning.

2.5.1 Formal Grammars

A grammar provides a set of rules which tell the parser how to form valid elements from the language's alphabet. The grammar ensures that the elements comply with the syntax of the language, but it does not describe meaning to these elements. A grammar will normally begin with a "start symbol", this defines where the parser should begin reading from.

Context-Free Grammar

A grammar is context-free if the left-hand side of each rule only contains a nontermi-nal symbol, the right hand side can be constructed of any symbols. If a language can be generated by a context-free grammar then it can be called a context-free language.

Regular Grammar

A regular grammar is similar to a context-free grammar in that the left-hand side of each rule still must only contain a nonterminal symbol, but the right-hand side is limited. The right-hand side can only be a single terminal symbol, a terminal symbol followed by a single nonterminal symbol or the empty string.

EBNF

Extended Backus–Naur Form (*EBNF*) is a code used to express context-free grammars for a formal language. It can use terminal and nonterminal symbols to make up the rules. A terminal symbol can be an alphanumeric character, a punctuation mark or a variety of other characters.

2.5.2 Types of Parsers

A parser has the job of determining how to read the input from the "start symbol" in the grammar. This can either be done with top-down parsing or bottom-up parsing.

Top-Down Parser

A parser is a top-down parser if it looks at the top level of the parse tree initially and then works down the tree according to the rules of the given grammar. It tries to find the left-most derivations of the input stream using a top-down expansion of the provided rules.

LL parsers are an example of top-down parsers. *LL(k)* is a variation of an *LL* parser which uses k tokens of lookahead, whereas an *LL(*)* parser is not restricted to a finite k lookahead tokens when parsing a sentence. An *LL(*)* parser instead makes decisions based on whether or not the following tokens are a part of a regular language (a language generated by a regular grammar).

Bottom-Up Parser

Bottom-up parsers begin with the stream-input and try to rewrite it up to the "start symbol". It tries to identify the input's lowest level basic elements first, then the elements containing those, etc. An example of a bottom-up parser is an LR parser.

Parser

3.1 Parser Generator

A parser generator is a tool which creates a parser from an inputted grammar. This is necessary so that we can parse the logic equivalences entered in to our application and create an internal tree representation.

3.1.1 ANTLR4

ANTLR is a widely used powerful parser generator that can be used to read, process, execute or translate structured text. It is used by a variety of companies such as *Twitter* for query parsing and *Hadoop* in data analysis systems. *ANTLR* supports a wide range of languages including Java making it a very suitable choice for our project.

From a grammar *ANTLR* generates a parser for that language and has the ability to automatically build parse trees representing the input. More interestingly, *ANTLR* also generates tree walkers which can be customised to visit the nodes of those trees and perform actions on those nodes. We use this functionality to build our own custom tree as described in Subsection 3.2.2.

ANTLR uses *EBNF* for its input grammar notation and uses an *LL(*)* parsing algorithm. This allows direct left recursion so that expressing our required syntax is easy and natural.

3.1.2 Alternatives to ANTLR4

ANTLR4 provides many benefits over previous versions and alternatives. *ANTLR4* uses a new variation of the *LL(*)* parsing algorithm that was used in *ANTLR3*. It uses an adaptive algorithm that is much stronger than the static grammar analysis algorithm used in *ANTLR3*. This means that *ANTLR4* will accept much more natural looking grammars and provides a lot more flexibility than in previous versions. The ability to write much more natural expression rules increases the usability as in previous versions it was often a struggle to write a grammar that *ANTLR* would

accept. *ANTLR4* aims to focus more on ease-of-use and chose simplicity over complexity where possible. This is a large reason for choosing this parser generator over the alternatives.

Although there are a few other tools that can generate source code similar to *ANTLR* that you can load into a debugger and step through, none of them have adaptive *LL(*)* parsers as *ANTLR4* does. Which means if we chose an alternative such as *JavaCC* we would need to configure our grammar to suit a weaker *LL(k)* parsing strategy. *JavaCC* is limited to the *LL(k)* class of grammars because it generates top-down parsers which means left recursion cannot be used. This would make generating the grammar more difficult and the result appear less natural.

3.2 Parser Generation

3.2.1 Grammar

The grammar we use with *ANTLR* can be seen in Listing 1. This grammar allows us to parse propositional and first order logic formulae.

```
grammar Expr;

prog: expr;

expr: '(' expr ')'                    #EXPR
    | '¬' expr                        #NOT
    | expr BINOP expr                 #BINOP_
    | (QUANTIFIER ATOM)* '[' expr ']' #QUANTIFIER_
    | expr '→' expr                   #IMPLIES
    | expr '↔' expr                   #IFF
    | TERM                            #TERM_
    | ATOM                            #ATOM_
    |                                 #ERROR
    ;

QUANTIFIER: ('∀' | '∃');
PREDICATE: ('A'..'Z');
BINOP: ('∧' | '∨');
TERM:   (PREDICATE ATOM*);
ATOM: ('a'..'z' | '⊤' | '⊥');
```

Listing 1: Expr.g4 grammar file for propositional and first order logic used by *ANTLR*

Labels are used to recognise when a rule has been entered and exited. *ANTLR* auto-

18

matically generates enter and exit functions for each rule based by their label. Some labels are followed by an underscore to differentiate them from the rules with the same name. An element in the grammar followed by a star (*) means that zero-or-more of these are allowed.

This grammar works for both propositional and first order logic formulae. This is because in propositional we will not allow the user to enter the first order symbols (quantifiers and terms) and in first order an error is returned if they try to use individual atoms without a predicate symbol.

Order of Precedence

Precedence rules can be used as a way of reducing the number of necessary parentheses. We use these in our grammar to decide which connective is the main connective when parsing a non-atomic formula. The precedence of propositional logical operators is defined in Table 3.1. The order of the grammar matches up with the order of precedence shown below, this is because *ANTLR4* will choose the first rule that matches and this is how we maintain precedence.

Operator	Precedence
$\neg$	1
$\wedge$	2
$\vee$	3
$\rightarrow$	4
$\leftrightarrow$	5

Table 3.1: Order of Precedence for Propositional Logic

In first order logic, quantifiers are evaluated after $\wedge$ and $\vee$ and before $\rightarrow$. Predicate symbols are evaluated at the same level as atoms.

3.2.2 Parser, Lexer and Walker

ANTLR automatically generates a parser and lexer for the grammar. The lexer tokenises the input source (the logical formula) into a token stream which can then be passed to the parser. A parse tree is generated by the parser which can be walked through by a custom walker.

The custom walker extends the *ANTLR* generated listener to override the methods for when the parser enters and exits each rule defined in the grammar. For example, when the parser enters the `NOT` rule, `enterNOT()` is called. Equally, when that rule is exited, `exitNOT()` is called. This allows us to walk through the parse tree and create our custom tree as we go.

An empty `FormationTree` is passed through to the walker alongside the parse tree so the walker can build the tree up as the parse tree is walked through. Each time a

19

grammar rule is entered the appropriate method (defined by the rule's label) in the walker is called to create that node in our tree. The structure of the tree is defined in more detail in Section 4.2.

3.2.3   Error Handling

ANTLR provides an `ANTLRErrorStrategy` interface for implementing a custom error handler to deal with syntax errors found during a parse by *ANTLR*-generated parsers. However, this has limited documentation and is a complex feature. Therefore, we have opted to relax the grammar by defining an error case so we can treat incorrect syntax as a semantic error instead. As shown in Listing 1 we define an `ERROR` case which accepts any incorrect syntax.

Once parsing is complete we walk over the tree using an `ExprWalker` (Subsection 3.2.2) and when an error case is entered we can set an error flag in the tree. This enables us to manage the error later by displaying an error message to the user without the application throwing up errors.

Internal Tree Representation

4.1 Compiler

The compiler combines all of the generated and custom parts of the parser. We define a `compile()` method to take the logic formula as input and parse it to create its tree representation. This is shown in Listing 2.

```
public FormationTree compile(String expr) {
    CharStream input = new ANTLRInputStream(expr);
    ExprLexer lexer = new ExprLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    ExprParser parser = new ExprParser(tokens);

    FormationTree tree = new FormationTree(null);
    ParseTree parseTree = parser.prog();

    ParseTreeWalker walker = new ParseTreeWalker();
    walker.walk(new ExprWalker(tree), parseTree);

    return tree;
}
```

Listing 2: `compile()` function to convert a logical formula into a FormationTree using the *ANTLR* generated lexer, parser and walker. The method `prog()` defines the where to begin the parse tree from in the grammar, (`prog` is the "start symbol" defined at the very top of the grammar).

4.2 Formation Tree

To build our internal representation of the logical formula we define a class `FormationTree`. This is a slightly modified binary tree that stores the root node and an

error flag as well as performing all of the tree operations required - such as adding a new node (detailed in Subsection 4.2.2).

4.2.1 Node Index

In the final application we manipulate individual nodes in the tree. If there are two nodes with the same values and subtrees we need to be sure we're applying a rule to the correct node, otherwise you could end up applying a rule to the first occurrence of that subtree instead of to the correct subtree. Therefore, it is essential that all nodes in the tree have a unique index.

[Key, Depth] Pairs

We represent this index as a [key, depth] pair. We define this pair using the three following principles:

Principle 1: A node's depth is defined as the level at which it is found in the tree.

Principle 2: A node's key is defined by its horizontal position in its level with the leftmost node beginning at 0 and increasing incrementally.

Principle 3: Incrementing the key also increments for any gaps in the tree where a node could be placed in the future. So the leftmost leaf will only have a key of 0 if it is the child of all the leftmost nodes in the tree.

For example, a formula $a \wedge b$ would only have three nodes - the 'and' connective ($\wedge$) would have a key and depth of 0 and the atoms a and b would be at depth 1. As a is the leftmost node it will be assigned a key value of 0, and b will be assigned 1. A larger example's $((x \wedge \neg y) \rightarrow ((x \vee y) \wedge u))$ key and depths are shown in Figure 4.1.

By **Principle 3** the incrementation of keys includes all gaps where it is possible for future nodes to be placed. In our example, the node y has the index $[2, 3]$ instead of $[0, 3]$ because if the left most x node had children they would use that index and each index must be unique.

Binary Representation of Index

The index of a node can be represented in binary to enable us to navigate around the tree easily. The depth represents how many bits should be read and the key defines the binary number. The depth is required so that two nodes with the same key cannot be read as the same.

In the example in Figure 4.1 '$\rightarrow$' and its left child '$\wedge$' both have the same key of 0, but their binary representations are different. '$\rightarrow$' does not have a representation as it is the root (length of binary read is zero) and $\wedge$ would be 0. You can see that each time you move down a level you add a 0 if you move left and a 1 if you move right. We go

Figure 4.1: Tree representation of $(x \wedge \neg y) \rightarrow ((x \vee y) \wedge u)$ showing the [key, depth] pair at each node and the binary representation of the index underneath.

into more detail about navigating the tree using the binary representation of indexes in Subsection 4.2.2.

Determining Child and Parent Indexes

Using this implementation of indexing not only guarantees that no two nodes will have the same index, but also we can easily work out the index of any child node, given we have the parent's index. Using the example, we have a subtree of $x \vee z$ at index [2, 2]. If we wanted to calculate the index of the child node x we perform the following actions:

1. **Key**: We perform a logical left-shift on the parent key (2 « 1 = 4 shown in Figure 4.2)

2. **Depth**: Increment the parent depth (2++ = 3)

So the resulting index of x is [4, 3]. To calculate the right child z's index we can simply increment the key of the left child using **Principle 2** to become [5, 3].

We can also find the parent node's index given a child node by performing the opposite operations. A logical right-shift finds the parent key and decrementing the depth finds the parent depth.
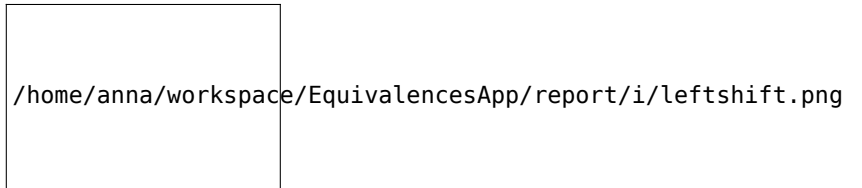
/home/anna/workspace/EquivalencesApp/report/i/leftshift.png

Figure 4.2: Left shift of 2 « 1 to give 4. MSB: Most significant bit, LSB: Least significant bit.

4.2.2 Tree Operations

Finding a Node

Finding a node based on its index is extremely straight forward using our node indexing scheme. We define a method `findNode()` which simply steps through the tree using the binary representation of the index as a map.

For each level in the tree starting at the root node we take the left most bit of the key and this tells us whether we should take the left or right child route. An unset bit and a set bit represent the left and right children respectively. After we have taken that path and moved to the next level we use a mask to remove the left most bit from the key so that it does not conflict with the next level's left most bit. Once the required depth is reached the current node is returned.

To find the node z defined by index [5, 3] in Figure 4.1 we would begin at the root node and take the left most bit of the binary representation to determine our path. The binary representation of this node is 101. As the left most bit of this is 1 we take the right path putting us at the right child node $\wedge$. The left most bit is then removed to leave 01, the next bit is 0 so we move left to $\vee$. Finally, the last bit is 1 so we move right giving us the node z. This is the at the required depth so the tree walk completes and returns the node.

Adding a New Node

We define a method `addNode()` to handle the addition of new nodes to the tree. The node passed in will already have its index set up so here we simply need to assign it to the correct position.

If the root of the tree does not exist then the node becomes the new root. Otherwise, we calculate the parent node's index as described in Subsection 4.2.1 and use `findNode()` to obtain the parent node that will already exist in the tree. We set this as the new node's parent.

Either the parent node is a unary operator and we set the child of that node to be our new node, or it is a binary operator and we must determine if the the new node is a

left or right child. If the node's key is divisible by 2 then the node is a left child, else it's a right child. We then set parent's child respectively.

4.3 Nodes

Due to the nature of logical formulae we need to be able to differentiate between unary operators, binary operators and atoms. We define a class `Node` which all of these will extend to define the core functionality required from every node.

Every node in the tree must contain a key, a depth, a value and a parent (unless it's the root in which case the parent is null).

We define `BinaryOperator`, `UnaryOperator` and `Atom` separately so we can set customised methods for each of them. Binary operators possess a left and right child, unary operators possess one child and atoms have none. The custom methods that require separate cases include a `clone()` method to duplicate the node (used in applying certain rules - Section 5.1), custom `toString()` methods and methods for determining truth values (Section 5.3).

Tree Manipulation

The application requires a few key features concerning tree manipulation: the ability to apply logical equivalence rules to a tree, compare the equivalence of two trees and randomly generate trees. This section will explain how these features were implemented.

5.1 The Rule Engine

To handle the application of logical equivalences we define a series of rules (Appendix A) that can be applied to individual nodes so long as they meet the required criteria for that rule. The `RuleEngine` manages the application and selection of these rules employing a `RuleApplicator` and `RuleSelector` respectively.

5.1.1 Rule Selection

The `RuleEngine` defines a method `getApplicableRules()` to compute which rules are applicable to an individual node in a tree. This returns a `BitSet` where each set bit represents a rule that can be applied.

Most of the rules can be put into categories (by operator) so only a subset of the rules need to be tested for a particular node. An example category is the 'and' connective ($\wedge$) - if the selected node is an 'and' connective then only the test for these rules should be performed as we know no other rules outside of that category will apply.

Within categories we define the tests for each rule individually. Some rules do not need any extra tests other than being assigned to a category. For instance the commutativity rule ($a \wedge b \equiv b \wedge a$) can be applied to any 'and' or 'or' operator node without any further tests being conducted. Whereas, the distributivity rule ($p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$) could be applied to an 'and' operator only so long as one of its children was an 'or' operator.

The `RuleSelector` defines methods of testing for the properties required by the rules such as testing for idempotence which proceeds by checking that both subtrees of the

node are equivalent. Each rule that can be applied to the current node has its index set in the returned `BitSet`.

5.1.2 Rule Application

When the user selects an applicable rule the `RuleApplicator` is used to apply that rule to the selected node. A function `applyRuleFromBitSet()` is defined in the `RuleEngine` to choose the correct manipulation to apply to the tree. It implements a large switch statement over the index of the rule to be applied with each case calling the necessary methods from the `RuleApplicator`.

To apply a rule the function creates new operators and sets the children appropriately according the rule applied. For example, when the distributive rule ($(a \vee b) \wedge (a \vee c) \equiv a \vee (b \wedge c)$) is applied the following steps are taken:

1. A new 'or' node is defined with its left child set arbitrarily to the left subtree at a and the right child set to a new 'and' node.

2. The left child of 'and' is set to the subtree at b and the right child is set to the subtree at c.

3. The parent of the selected node has the selected child set to the new 'or' node and `relabelNode()` is called to assign new indices to the moved nodes.

In the case where this rule would be applied backwards and two copies of a are required, we define a custom `clone()` method to duplicate the subtree so no two nodes reference the same child.

Some rules can require user input, for example in $\perp \equiv a \wedge \perp$ we require some input to decide which variable to use in the place of a. The selected variable can be passed in when applying the rule from `BitSet`.

5.2 Generating Equivalent Formulae

It is possible for a user to enter formulae manually, however, we also want to provide a quick option to automatically generate usable equivalent formulae.

5.2.1 Random Formula Generation

When generating a logical formula we should be able to specify certain parameters such as how many variables and connectives are used. These parameters are used to generate different complexities of formulae for different difficulties.

The `Compiler` is used to generate a random formula. The function `generateRandomEquivalence()` takes the number of variables we want to use and the maximum depth of the tree to be generated. A selection of variables that can be used is then

chosen (Subsection 5.2.2) and passed through to a function `generateSubEquivalence()`. This recursively generates subformulae until the end criteria are fulfilled and returns the string representation of the formula. The string representation is returned so that when the formula is compiled it will be assigned the correct indices.

The end criteria are either the maximum depth or a random criterion has been reached. At each level the random criterion decides whether or not each node should have children so that we don't always get a balanced tree. This provides us with more interesting equivalences to work with. Currently the random criterion has a $\frac{1}{5}$ chance of deciding to not generate children for a node because this produces nice looking results - trees which are generally not completely imbalanced but have a good level of variation to them.

During each recursion if the end criteria has been met then we can randomly choose an operator and variables to assign to it and set this as a final subformula. Else, instead of assigning variables we call the function again and assign the returned results to the operator.

Random operators and variables are selected using instances of `Random` from their respective lists. All the operators and a reduced selection of variables are available for use in the formula.

5.2.2 Variables

Initially all the variables would appear in a generated formula with the same probability. However, this proved to be a poor approach because you often ended up with many truth values ($\top$ and $\bot$) in the formula. This made the equivalences very easy to solve as you could usually just reduce both to their equivalent truth value. Proving the formulae were equivalent this way doesn't promote the users learning experience as they will repeatedly only use a very small subset of basic equivalences such as $a \lor \top \equiv \top$.

The solution to this problem was to assign probabilities to the variables. We define an enum `Variable` (see Listing 3) where all the entries have a value and a weight. The weight is defined on a scale of 0 to 100 with the higher weights being more likely to be chosen.

Initially all the atoms apart from the truth values are defined with a weight of 100 and the truth values are defined with a weight of 50 (so they will appear half as regularly as other variables). The application also provides the ability for a user to choose the probability they want these truth value atoms ('$\top$' and '$\bot$') to appear - to either increase or decrease the difficulty. When the user changes this probability the application will update the weights for the truth value variables.

The enum provides the method `randomVariable()` to return a random variable based on their weights. A simple analogy for this method is to imagine a ruler where the maximum value is the sum of the weights and each weight is dashed on the ruler. If you randomly placed your finger somewhere along the ruler then the segment you

land on becomes the variable you return. So variables with smaller weights are less likely to be selected.

```java
public enum Variable {
    A("a", 100), B("b", 100), C("c", 100), D("d", 100),
        E("e", 100), F("f", 100), G("g", 100), H("h", 100),
        I("i", 100), TRUTH("⊤", 50), FALSE("⊥", 50);


    ...
}
```

Listing 3: Variable enum defining values and weights for variables

5.2.3 Random Rule Application

Applying random rules to a formula is used in generating an equivalent formula. When only one logical formula has been specified the user can request an automatically generated equivalence.

The second formula can be generated easily by repeated application of random rules. This guarantees that the generated formula will be equivalent to the first (because only applicable rules resulting in equivalence will be applied) and also allows us to set the difficulty of the proof based on how many steps there are between the two formulae.

There are two sections to the rule applications: choosing a random node and choosing a random rule.

In our tree we define a method `randomNode()` to randomly select a node. This method calculates the maximum depth of the tree by walking through the tree and returning the maximum depth found. A random level of the tree is chosen using an instance of `Random` and the maximum depth before walking down the tree to that level - at each level the method arbitrarily picks a child to follow. When the selected random level is reached the current node is returned.

Once a node has been selected to perform a rule on we call `getApplicableRules()` (Subsection 5.1.1) to obtain the `BitSet` of applicable rules. From this the number of applicable rules is used with another instance of `Random` to select a rule to apply. Some of the rules require user input, so we define a special case for these rules to randomly select a variable to be passed through to `applyRuleFromBitSet()` (Subsection 5.1.2) when applying the rule. If no user input is required then we pass through null.

5.3 Truth Tables

For a user to be able to complete a proof of their own custom equivalences we need to ensure that they are actually equivalent and the proof is completable before they begin. To do this we can use truth tables (Section 2.3).

5.3.1 Generating Truth Tables

`TruthTable` contains a `SortedSet` of variables, a table with the type `LinkedHash-Set<ArrayList<Integer>>` (each `ArrayList` represents a row in the table) and a string formula. They can be printed out as shown in Table 5.1, we will use this representation of our truth tables at explain each stage to help with the explanation.

To generate the truth table we need to generate the values for the atoms (the first two columns of the example shown in Table 5.1) then apply them to the formula to find the final column's truth values.

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

Table 5.1: A printed out example of a generated truth table

We use a `SortedSet` to store the variables in so that when comparing two truth tables the columns will be in the same order to make comparisons easier. A `LinkedHashSet` is required for the table because it has a predictable iteration order which is required for comparisons of tables. The same argument is used for using an `ArrayList` for each row. This provides us with an ordered collection where we have precise control over where each element is inserted, enabling us to have easy access to the values in the rows - necessary for extending a table in certain cases.

Creating Input Truth Values For Atoms

For creating the first two columns of the table shown in Table 5.1 we define a function `createAtomicTruthValues()`. This takes the number of variables present in the formula (in this case the number of variables is 2 - a and b) and calculates the number of rows that will be present in the table by squaring this number (giving us 4 rows).

A new `LinkedHashSet` table is created and for each row in the table and each variable we add a new entry as shown in Listing 4.

This produces a `LinkedHashSet` which can be represented by the table shown in Table 5.2. A similar result is produced for any number of variables.

```
for (int i = 0; i < rows; ++i) {
    ArrayList<Integer> row = new ArrayList<Integer>();

    for (int j = numVars - 1; j >= 0; --j)
        row.add((i / (int) Math.pow(2, j)) % 2);

    table.add(row);
}
```

Listing 4: Filling in table for atomic truth values

a	b
0	0
0	1
1	0
1	1

Table 5.2: The result of `createAtomicTruthValues()`

Determining Truth Value For The Tree

Each row of the table is used to evaluate the truth values of the formula for the final column in the table. We define the function `createTruthTable()` to create the final truth table for the tree of a logical formula. This calculates the number of variables by walking through the tree and passes this value to `createAtomicTruthValues()` to generate the atomic truth values required as described above.

Looping through each row of the table allows us to compute the corresponding truth value for the formula. To compute the formula's truth value for that row a `HashMap` of `String` and `Integer`s is created to map each variable to a truth value. The `FormationTree` defines a method `getTruthValue()` to calculate its truth value using this map.

The tree determines its truth value by recursively calling `getTruthValue()` on its nodes beginning at the root. Using the example shown in Table 5.1 for $a \wedge b$: the root node is '$\wedge$' so the result would be a logical and of the left and right children. When `getTruthValue()` is called on an atomic value it either returns 0 or 1 if the value is '$\bot$' or '$\top$' respectively, else it returns the value defined for that atom in the map.

When evaluating the truth value for a formula the application uses shortcuts to reduce the amount of computation required. For example, when computing the truth value for the first row of our table [0, 0], where a and b both evaluate to 0 (or '$\bot$'), once the application knows the left child a is 0 there is no need to evaluate the right hand child because it's impossible for the overall truth value to be 1 (or '$\top$'). Therefore, the right hand evaluation is skipped.

The returned truth value is added to the end of its corresponding row in the table.

Now the table is complete and can be represented as shown in Table 5.1.

The other operators use return their truth value using similar methods. The truth tables for all the used logical operators are defined in Table 5.3.

a	b	a∧b	a∨b	a→b	a↔b	¬a
1	1	1	1	1	1	0
1	0	0	1	0	0	0
0	1	0	1	1	0	1
0	0	0	0	1	1	1

Table 5.3: Truth table for all binary and unary operators

5.3.2 Testing Equivalence of Tables

`TruthTable` defines a method `testEquivalence()` which compares its own truth table to another. There are three cases that could be possible when comparing truth tables:

Case 1: Both formulae have the same variables and can be proven equivalent;

Case 2: Both formulae have different sets of variables and can be proven equivalent;

Case 3: The formulae can not be proven not equivalent.

Checking Tables Are Equal

In the **Case 1** we could have two formulae such as $a \land b$ and $b \land a$. These have contain identical sets of variables. They are equivalent if they contain the same set of variables and their truth tables are equal - identical apart from the result formula.

Tables are checked for equivalence using `tablesEqual()`. If the two tables to be compared aren't the same size the method immediately returns false to reduce computation time. If they're the same size it loops through each row in both tables and checks that they're equal. If all the rows are contained in each table the method returns true.

Different Variables and Differently Sized Tables

The variable sets need to be compared in the previous case because the tables can be equivalent even if they use different variables. In this case the proof may or may not be solvable. For example, $a \land b$ has an equivalent table to $p \land q$ but we do not define any rules for replacing one atom in the tree with another so you would not be able to complete the proof. Whereas, $a \land \neg a$ and $b \land \neg b$ have equivalent tables and can be proven equivalent because they both reduce to '$\top$'.

However, a basic check to ensure the two variable sets are equal does not always work as there are cases where the variable sets will be different as described above.

Case 2 defines a case where both formulae are equivalent but have different variable sets. Another example of this would be $a \wedge \top$ and $\bot \vee a$ which both contain an atom not present in the other formula.

In this case we extend each table to include the missing variables (so both variable sets become equal) and then execute another comparison using `testEquivalence()` where the tables will either fall into **Case 1** and be equivalent or **Case 3** and not be equivalent.

Due to the rules such as the absorption rule $(a \vee (a \wedge b) \equiv a)$ it is possible for formulae with differently sized variable sets to be equivalent too. These equivalences can also be shown to have equivalent truth tables by table extension.

Table Extension

In this section we will be using $a \wedge \neg a$ and $b \wedge \neg b$ as our example formulae. The initial truth tables for each of these formula are shown in Table 5.4.

a	$a \wedge \neg a$
0	1
1	1

b	$b \wedge \neg b$
0	1
1	1

Table 5.4: Truth tables for $a \wedge \neg a$ and $b \wedge \neg b$

To be able to compare the truth tables of these formulae we need to extend them to include all the variables that appear in both formulae, the resulting tables should be able to be represented as shown in Table 5.5.

First, a new variable set is created and the variables from both formulae are added to it. In our example this set would become $[a, b]$. Two new extended tables are created, the first initialised to the first original table and the second to the second original table. The function iterates through the new variable set and if a variable does not appear in one of the original sets it is added to its respective extended table using `addVariable()` as shown in Listing 5.

Our example would need to extend both tables once until they had identical variable sets.

Adding a new variable to a table is done by duplicating each row and on each setting the value of the new variable to 0 and 1 respectively. The row $[0, 1]$ in the formula $a \wedge \neg a$ would need a b variable added so would become two rows of $[0, 0, 1]$ and $[0, 1, 1]$ where the new variable has been inserted into the second column. The same would occur for the second row $[1, 1]$.

Once both tables have been extended the function calls `tablesEqual()` on them to confirm that they have the same truth values and are equivalent.

```
for (String var : all) {
    if (!variables.contains(var))
        extendedTable1 = extendedTable1.addVariable(var);
    if (!v2.contains(var))
        extendedTable2 = extendedTable2.addVariable(var);
}
```

Listing 5: Extending the tables by adding new variables

a	b	$a \wedge \neg a$
0	0	1
0	1	1
1	0	1
1	1	1

a	b	$b \wedge \neg b$
0	0	1
0	1	1
1	0	1
1	1	1

Table 5.5: Extended truth tables for $a \wedge \neg a$ and $b \wedge \neg b$

If this check also fails then the formulae are **Case 3** where they are not equivalent and the method returns false.

Android VS Web Applications

6.1   Which Platform?

In an ideal world it would have been very interesting and useful to have been able to build our application for multiple platforms - including smartphones, tablets and the web. However, for this project it was obvious that focusing on one platform would provide us with a much more complete and robust final project.

The options were narrowed down by the fact we wanted to display the formation tree of the current formula as this would require a fairly large amount of screen space. Therefore, we opted not to target the application for a smartphone. If this were to be an extension we could simply remove the formation tree functionality to make the application usable on the smaller screen. This left us with the option of targeting a tablet or a website.

If we chose to build an app for a tablet, personal preference would cause us to choose to develop for Android. This is because I had not had any experience programming with Android before and thought it would be a good skill to learn, whereas I have programmed for iOS in the past. However, we still needed to decide if we would develop for an Android tablet or the web.

6.1.1   Tablet Apps VS Web Apps

User Interaction

The biggest difference between an application on a tablet and on a computer is the difference in input methods. On a tablet a user uses gestures to interact with objects on the screen whereas on a computer a user can use the mouse to perform a variety of actions.

This difference seriously affects the way the user interacts with the application. A tablet application allows a more interactive application that would rely a lot more on user interaction such as using swiping gestures to move around the screen and selecting specific formula to undo back to that formula. This is because unlike on a

computer a tablet does not have the ability to perform different types of clicks (left or right on the mouse) so the interaction needs to be well executed to be intuitive and easy-to-use. Therefore, choreographing the gestures and interaction on a tablet is more difficult than on a web app.

Portability

Tablets are much easier to use discretely on the move or in lectures than a laptop. As we are creating this application with the intention of it being a learning tool, the ability to use it easily and to be unintrusive and undisruptive in a lecture is a major bonus. A web app could still be run on a laptop in this environment but they are generally considered to be more disruptive.

While a web application could be used on a tablet the user experience is much richer with native applications. The graphics and user interface used for web apps are not as smooth as on a native app. This is because the graphics for a native app are already installed and therefore download speed during use does not need to be considered while developing[12].

Building a native app means that the application can utilize all the built in hardware on the phone and access internal data. This functionality is not available through a web app.

Availability

There are currently no Android applications that provide the functionality we have detailed so far (Section 2.4). Whereas there are more complex options available on a computer such as Pandora (Subsection 2.4.2), and despite the fact they do not provide the same functionality as our application they are quite useful logic applications.

Learning Experience

As I have not programmed any Android applications in the past I had a personal preference to create an Android application. This would provide me with more to learn from the project unlike a web application which I have had a large amount of experience with.

Decision

In conclusion we decided to develop the application on an Android tablet device. Despite this being the more complex solution we believe that the application is more suitable to being used on a tablet and would be more likely to be used by students.

Android

Based on the Linux kernel Android is an operating system primarily designed for touchscreen mobile and tablet devices[3]. The project responsible is the Android Open Source Project (AOSP) and is primarily lead by Google. The system provides a large user interface library and supports its graphics using the OpenGL-ES standard[4].

7.1   Technical Background

7.1.1   Android Platform Components

The Android system can be divided into four main levels (listed from top to bottom): applications, the application framework, libraries and runtime, and the Linux kernel.

There are many default top level applications such as Browser, Camera and Music. The application framework is an API allowing high-level interactions from applications to the Android system. Android libraries are used for many standard functions like web browsing and rendering graphics in the application framework and in the Dalvik runtime (Dalvik is the process virtual machine in Android). The Linux kernel acts as a communication layer for the underlying hardware.

When developing for Android a developer will only tend to work on the top two layers.

7.1.2   Applications

Applications extend the functionality of Android devices and are primarily developed in Java using the Android SDK. The SDK provides an extensive set of development tools and there is plenty of documentation available.

Users of an Android device can download third party applications through an app store such as *Google Play*. There is a growing number of third party applications and the app stores allow a user to browse, download and update any applications published suitable for their device.

There were more than one million applications available for download from the Google Play Store as of July 2013[11].

7.1.3 Power Management

As with most mobile and tablet devices most Android devices are battery powered. Unlike desktop operating systems such as Windows, Android is designed to keep power consumption low. This is because desktop operating systems can make the assumption that they will be connected to unlimited mains electricity.

To keep power consumption at a minimum Android will set apps to idle when they are no longer in use, suspending them so they don't consume battery or processing power. Keeping apps idle means that they do not need to be closed and reopened when a user wants to access them again which also helps reduce power consumption.

7.1.4 Market Share

Android's main competitor in the mobile platform market is Apple's iOS. In research conducted in the fourth quarter of 2012, *Kantar Worldpanel Comtech* showed sales of all Android smartphones worldwide outpacing the iPhone by a huge margin: 70 percent to 21 percent of the smartphone market[6].

In the tablet market the iPad dominated Android's 7" tablets with 53.8 percent to 42.7, which is lower than Android's smartphone market but steadily increasing[5].

7.2 Features

7.2.1 Graphics and Widget Library

Android does not use the Swing library or the Abstract Window Toolkit (AWT) UI libraries used in most Java applications. Instead the graphical user interface for an Android app is built using a hierarchy of `View` and `ViewGroup` objects. The framework employed is similar to Swing but based around `View` objects rather than `JComponent` objects.

`ViewGroup` objects are hidden containers used to define how the child views are positioned. Buttons and text fields are some examples of `View` objects, they are UI widgets that manage what a user can see and do on the screen. `View` and `ViewGroup` components can either be defined in XML or programmatically.

Widgets require the application's `Context` to be provided at creation. The `Context` of an application provides global information about its environment. As the name suggests, its provides the context of the current state of the application allowing newly

created objects to understand what has occurred. It can be used to obtain information about another part of the program such as another `Activity`.

7.2.2 Activities

An `Activity` is the base for the user interface in Android, it represents the presentation layer which a user will see. Android applications tend to have multiple activities which are switched between during runtime.

Often activities will be presented as a full-screen window but they can also be used as floating windows or embedded inside other activities (implementing `ActivityGroup`).█ All `Activity` classes must be declared in the Android manifest file. This is an XML file that holds all the essential information the application requires before it can run any code.

When a new `Activity` begins it is pushed onto the "back stack" and the previous `Activity` is stopped. As the stack has the standard "last in, first out" structure when a user presses the back button the current `Activity` will be popped off and the previous `Activity` is resumed.

7.2.3 Fragments

A `Fragment` is a piece of an application's UI that can be placed inside an `Activity`. Each `Fragment` defines its own lifecycle (each with its own methods: `onCreate()`, `onPause()`, etc.). Although the lifecycle is reliant on its `Activity` (if the `Activity` is stopped then no fragments inside the `Activity` may be started) a `Fragment` is able to respond independently to events.

`Fragments` allow for modularisation of the layout code so that the user interface is more easily adjustable.

7.2.4 Intents

`Intents` are used as a link between two activities for transferring data between them. Each `Intent` is an asynchronous message suppling an action and data to the new `Activity`. They are most often used for starting another `Activity` and providing that `Activity` with whatever information it requires from the previous. Intents are extremely useful as they allow the creation of a loose binding between two separate components.

7.2.5 Layouts

The layout is a view group which defines the visual structure in an `Activity` or widget. A layout can be declared in an XML `Activity` or `Fragment` file or the layout

elements can be instantiated at runtime. We can view a visual representation of the layout of an XML file in the graphical layout editor in Eclipse thanks to the ADT plugin. Declaring the layout in the XML file allows for better seperation of the presentation and behaviour of the app. Different XML layouts can be defined for different screen sizes or screen orientations.

There are a few different types of layout that can be used including `LinearLayout` and `RelativeLayout`. `LinearLayout` aligns all of its children either vertically or horizontally and `RelativeLayout` displays its child views in their relative positions. There are other types of layout that will be described as they are used in the application.

Every layout requires layout parameters, these control how the layout will appear on the screen. As layouts are view groups they can be nested inside each other. Defining layout parameters allow us to control how much of the screen each of these layouts take up.

7.2.6 Multi-Touch Gestures

The user interface is based around direct manipulation. This means using touch inputs that are intended to correspond to real-world actions, for example swiping, pinching and tapping to control on-screen events. The response to user actions is designed to be immediate and provides a fluid touch interface. The device often provides haptic feedback (forces, vibrations or motions) to the user using the vibration capabilities.

Android takes advantage of internal hardware such as accelerometers, gyroscopes and proximity sensors in some applications to respond to other user actions, such as re-orientating the screen from portrait to landscape.

7.3 Difficulties Developing For Android

During development of the application we encountered some difficulties specific to Android which we would not have experienced if we had developed a web application. There are detailed below and will be referencing aspects of the project detailed later in the report.

7.3.1 Screen size and Scrolling

Tablets have a limited screen size compared to a computer screen. Although we have established that the screen size would be suitable for displaying the formation trees of the equivalences it does present the difficulty of moving around the application. Android provides a scrollable view that allows the view to scroll either horizontally or vertically, but not both. Nested scrollable views are not allowed as the application would not be able to understand which view you were trying to scroll through. This

is problematic as it is possible for a large formation tree to be larger than the width of the screen. We could not set the scrollable view to scroll horizontally because we require vertical scrolling for long equivalence proofs. On a web app this would be an easy solution as nested scroll boxes are easy to implement. To combat this problem we provided the user with buttons to move the tree left and right if it is larger than the size of the screen.

7.3.2 External Libraries

Using external libraries with Android proved to be much more of a challenge than expected. This was especially evident when looking for an external library used for drawing attractive trees. Most of the libraries available hooked straight into Java's Swing components and so huge amounts of the library would need to be rewritten for it to function on Android.

This meant that any external libraries used needed to separate the core functionality from the actual rendering to be usable. Even then some tweaking was required to get them to function correctly. When using Abego TreeLayout (Subsection 8.7.6) we had trouble importing it into the project and even though it separated out the rendering and the functionality we needed to alter some of the underlying code. This is because it used features from Java's AWT and they had to be replaced by the Android equivalents.

7.3.3 Custom Features

Android provides the ability to generate your own custom features when the ones you want are not available. Such as the custom keyboard described in Section 8.4. These features would not need to be implemented on a web app as the computer would already provide all of them. Implementing the custom keyboard was a difficult process as the `KeyboardView` we needed to extend is not well known. Although there is documentation on the `KeyboardView` there are not many examples and there are some issues with implementing it.

Designing our own `KeyboardView` has the downside that it is a `View` and as such is a part of the layout in an Activity. This means the `View` will obstruct all the layouts implementing it but we are in complete control of our application, keyboard included.

`KeyboardView` is not part of any automatically found package meaning the full path to the class is required. Therefore, the graphical layout editor (which shows a graphical representation of the XML file) cannot show a representation of the keyboard making the design process slightly more difficult.

One of the most complex issues was making the keyboard appear at the right time, and even more difficult was making the standard keyboard invisible at the same time. Android is very persistant in having the standard keyboard appear. The workaround was to set the input type of the text box as `TYPE_NULL` (telling Android we wouldn't

be editting that text box), however this hides the cursor meaning we needed to add our own cursor keys to our keyboard. This is currently considered a bug by some users[7].

The App

8.1 Activities

In this application there are three activities:

- `MainActivity`
- `EnterEquivalenceActivity`
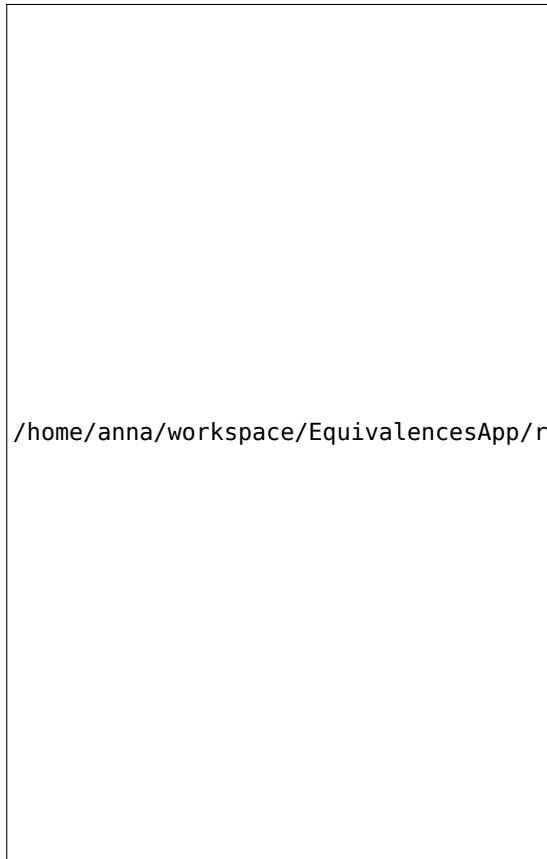- `BeginEquivalenceActivity`

8.1.1 Main Activity

`MainActivity` controls the first screen the user sees when they start up the app. This allows the user to change the settings (difficulty and probability of truth values) and choose either propositional or first order logic, shown in Figure 8.1.

The fragment (Subsection 7.2.3) XML file `fragment_main` contains the widgets you can see in Figure ... These are defined as `TextView`, `Button`, `SeekBar` and `Spinner` widgets. `TextView` and `Button` widgets are self-evident, the `Spinner` is used to create the dropdown menu to select the difficulty and the `SeekBar` is used for the slider that controls the probability of truth atoms appearing in the generated formula.

When the "Propositional Logic" or "First Order Logic" button is clicked the function `startPropositional()` or `startFirstOrder()` is called respectively. Both of these functions call `startLogic()` and simply pass in boolean parameter defining if the equivalences should be first order or not. This creates an `Intent` (Subsection 7.2.4) to start a new `EnterEquivalenceActivity`. The difficulty (Section 8.2) and selected probability (Subsection 5.2.2) of truth values are added to the intent and it starts the new activity.

8.1.2 Enter Equivalences Activity

`EnterEquivalencesActivity` allows the user to either manually enter or randomly generate the formulae to prove equivalent. As shown in Figure 8.2 the XML fragment

Figure 8.1: `MainActivity` allows a user to change the settings and choose either propositional or first order logic.

`fragment_enter_equivalences` defines two `EditText` widgets to manually enter the formulae. These widgets use a custom keyboard for entering logical symbols and variables. Two "Randomise" buttons are defined to automatically generate equivalences.

Upon the "Start" button being pressed the activity will check the two formulae are equivalent. If there is a problem with the formulae entered (either they are not equivalent or they have a syntax error) then the appropriate error message will be shown. If both the formulae are equivalent a new `Intent` will be generated to begin a new `BeginEquivalenceActivity`. The intent will carry the two formulae to the new activity so that the user can begin the proof.
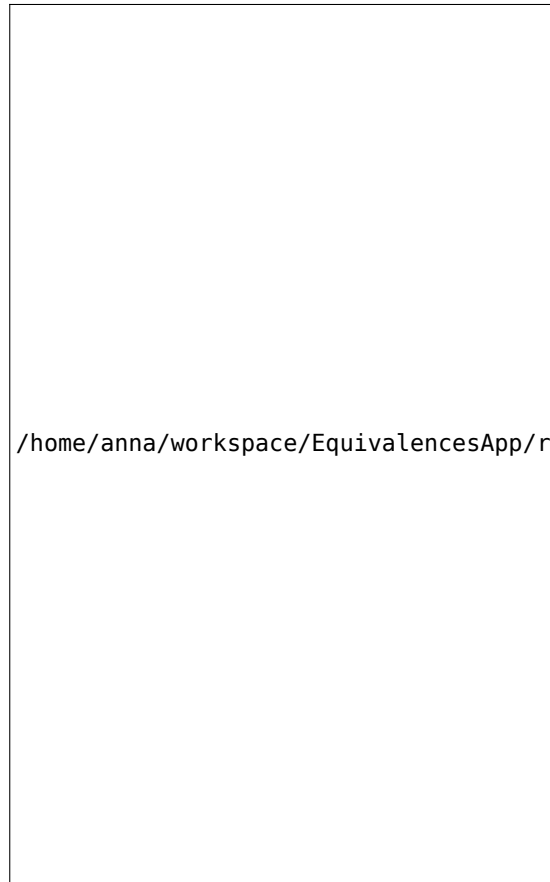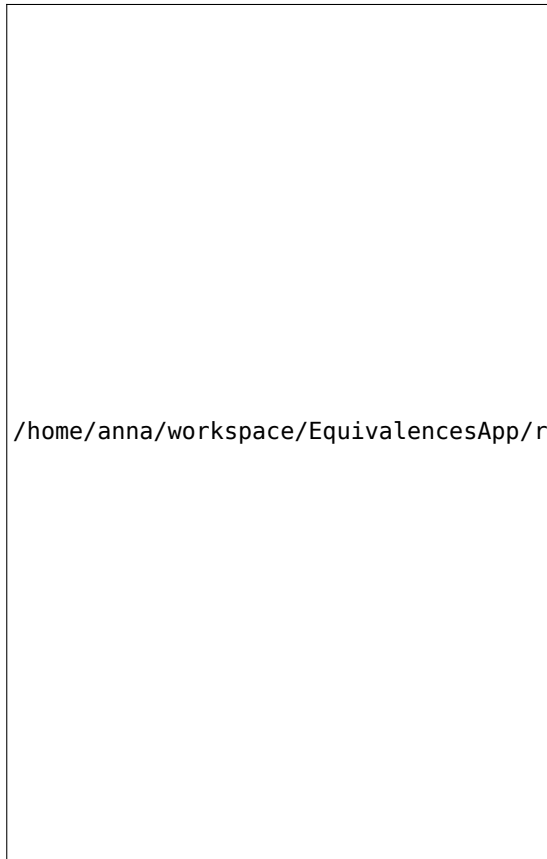
/home/anna/workspace/EquivalencesApp/report/i/screenshots/PropositionalEnt

Figure 8.2: On selecting propositional logic `EnterEquivalencesActivity` allows a user to enter propositional formulae.

8.1.3 Begin Equivalence Activity

The XML file `fragment_begin_equivalence` employs a `ScrollView` as its outer layout, this is a specific type of view group extending `FrameLayout`. A `FrameLayout` surrounds one child (usually another layout) containing its entire contents, therefore `ScrollView` allows us to scroll through the content so the layout can be bigger than the physical display.

A `LinearLayout` (Section 7.2.5) is used as the child of the `ScrollView` which then contains all the widgets required for completing the proof. This layout consists of two main sections, one for each formula. The start formula is placed at the top of the layout working down and the end formula is placed at the bottom working up. Each end displays its current formation tree (Section 8.7) and allows the user to undo or redo rules (Section 8.9). This display is shown in Figure 8.3.

/home/anna/workspace/EquivalencesApp/report/i/screenshots/Example1-2.png

Figure 8.3: `BeginEquivalenceActivity` shows the formation trees of the entered equivalences.

8.2 Difficulty

The selected difficulty only affects the proof when formulae are randomly generated because when a user manually enters their own formulae they are in complete control over what they want to prove.

8.2.1 Difficulties

The user can choose a difficulty from either beginner, intermediate or hard in the settings of the application. Each of the difficulties have set values for their parameters which are used in generating formulae.

8.2.2 Parameters

The parameters used in generating equivalent formulae are the number of variables that can appear in the inital formula (`numVars`), the maximum depth of the formula's tree (`depth`) and the number of rules we will apply to the initial formula to generate two formulae equivalent to the original (`numRules`). These are shown in Table 8.1.

Difficulty	numVars	depth	numRules
Beginner	3	2	3
Intermediate	5	3	4
Hard	10	4	5

Table 8.1: Parameters used for each difficulty to generate the initial formula

8.2.3 Probability of Truth Values

Initially the probability of the truth values was tied to the difficulty the user selected, however it was suggested that it could be beneficial to the user to have direct control over this parameter.

As described in Subsection 5.2.2 the user has the ability to set the likelihood of the truth values being chosen (up to the same amount as other variables). The user can set this value using the `SeekBar` in `MainActivity`. The `SeekBar` has a maximum value of 100 (so at maximum it will be equally likely to be chosen as other variables) and a default value of 50 (half as likely). This value is passed through with the other parameters when generating new formulae.

8.3 Random Generation of Equivalences

`EnterEquivalencesActivity` has a compiler (Section 4.1) and a rule engine (Section 5.1) which is used to generate formulae and apply rules.

The top and bottom "Randomise" buttons call the functions `randomiseStart()` and `randomiseEnd()` respectively. These functions simply call the `randomise()` function and pass through which text box it is trying to generate a random formula for.

There are three possible cases when one of these buttons have been clicked:

Case 1: Neither text box contain formulae and one needs to be generated from scratch.

Case 2: The other text box contains a valid formula and an equivalent one needs to be generated.

Case 3: The other text box contains an invalid formula and an error will be thrown.

The `randomise()` function takes both text boxes as parameters, the first being the text box we want to display the generated formula in and the second the other text box. It then determines if the other text box is empty and acts accordingly.

8.3.1 The Initial Formula

The `Compiler` is used to generate a random formula when a "Randomise" button is clicked (Subsection 5.2.1) using the user specified parameters.

Previously, this generated formula would be set as the new formula in the text box corresponding to the button clicked. This gave an unfortunate side effect when a second equivalence was generated (**Case 2**), where it would usually be longer than the first - due to the fact that there are more rules which increase the size of a formula.

This is not a problem if the user begins the proof immediately, but if the user clicks the first randomise button again the app would generate a formula equivalent to the previously generated longer formula creating a longer again formula. Repeating this by alternating clicks on each button resulted in the formulas continuously increasing in length disregarding the difficulty set.

The solution to this was to set the generated formula as an initial formula to which rules would be applied to generate any required formulae.

8.3.2 Generating Equivalent Formulae

In **Case 1** the other text box has nothing in it so a new formula needs to be generated by the compiler. If there is already something in the current text box it will be over-written. The initial formula is remembered by the activity, compiled into a tree and a number of random rules are applied to the tree (depending on the difficulty). This new tree's formula is set as the first formula in the text box.

In **Case 2** there are two sub cases. Either the other formula was generated by the activity and an initial formula has been set, or it was manually inputted and there is no initial formula. If no initial formula is set then it is set to the manually inputted formula in the other text box.

The initial formula is then compiled into a tree and has random rules applied to it (as above in **Case 1**). This newly generated tree has its formula set to the textbox.

Case 3 occurs when the initial formula has been set from a manually inputted formula with incorrect syntax. In this case the parsing of the formula will fail and an error flag will be set. When this flag is set an error message will be shown informing the user of their error and no equivalent formula will be generated.

8.4   Custom Keyboard

In the application we enter formulae in alpha numeric fields. Android provides context sensitive soft keyboards for entering text so different keyboard appear in different situations. However, it does not provide a keyboard for logic symbols meaning a different keyboard is required.

Other than the standard keyboards provided by Android there are manufacturer keyboards and user installed keyboards. The manufacturer keyboards are a more simple reduced version of the Android keyboards also without logic symbols. User installed keyboards allow a developer to have complete control over the keyboard but end users of the application would have to also install the keyboard service. Installing a keyboard service would be an inconvenience for the user and also presents a security risk because everything that passes through the keyboard can be tracked by the application, such as sensitive information.

8.4.1   Keyboard View

Android has a `KeyboardView` (difficulties with using this described in Subsection 7.3.3), which is a view that allows us to define the keys the keyboard displays. The keyboard is made up of rows of keys. Each key is associated with a label and a code. The label defines what is shown on the key button and the code is passed to the app when the key is pressed. If a code is not defined it is generated from the unicode value of the label.

Two keyboards are defined, one for propositional logic and one with extra keys for first order logic (Figure 8.4). The keys are defined in XML files `logickbd.xml` and `firstorderkbd.xml`, each time a key is pressed the key's code is passed to the `CustomKeyboard` class. A key listener is used in `CustomKeyboard` to detect when a key is pressed, finds the view in focus and ensures it's an `EditText` (else it aborts) then executes an action based on the key's code.

8.4.2   Keys

If a variable or symbol key is pressed then the corresponding symbol is inserted into the text box. Undo deletes the last value and clear deletes everything in the text box. Hide is used to hide the keyboard and the arrow keys are used to move the cursor left and right. The problems with the cursor and hiding the keyboard are detailed in Subsection 7.3.3.

8.4.3   Registering Text Boxes

Each text box that requires the logic keyboard needs to be set up to use it. The function `registerEditText()` enables the application to register multiple text boxes to use
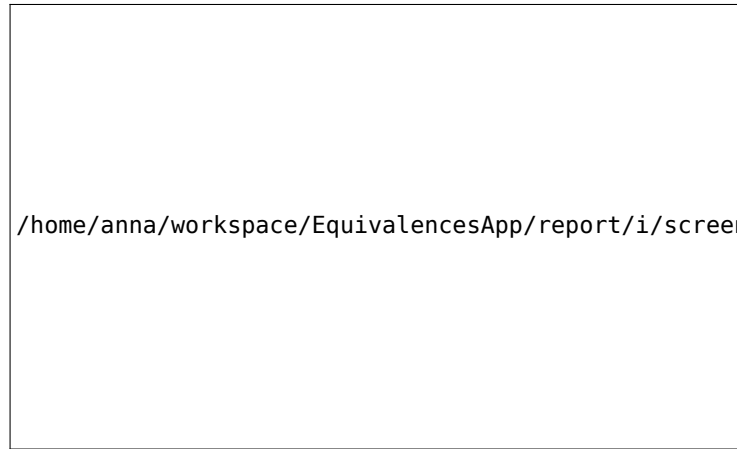
Figure 8.4: First Order logic keyboard generated by `firstorderkbd.xml`.

the keyboard.

First this function sets the text box's listeners to show the custom keyboard when the text box is selected and has focus. It also hides the standard keyboard, sets up the new cursor and turns spell check off.

8.5 Truth Tables

After two formulae are entered (either manually or generated) the user can click the "Start" button. This button is linked to `startEquivalence()` which uses truth tables to check the formulae are equivalent and creates an intent to start a new activity.

8.5.1 Equal Formulae

The function `startEquivalence()` finds the `EditText` boxes holding the formulae by their ids defined in the XML file and retrieves the text within them. These two formulae are both compiled into their `FormationTree` and a truth table (Section 2.3) is generated for each tree.

If these truth tables are equivalent then a new intent is created to carry the equivalences to the new `BeginEquivalenceActivity`. Before the formulae are added to the intent we set the equivalences to the string representation of their respective trees. This is to deal with the case where no brackets have been used and have been implied by the parser. Setting the formulae to the string representation of the tree adds in the brackets where they have been implied to avoid confusion later on. The two equivalences can then be added to the intent and the new activity is started. This is shown in Figure 8.5 and Figure 8.6.
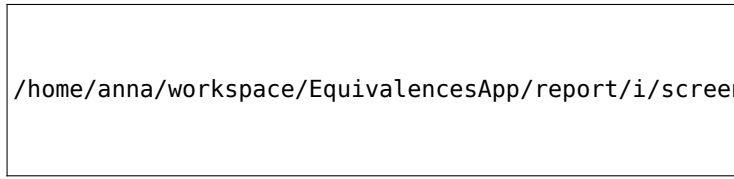
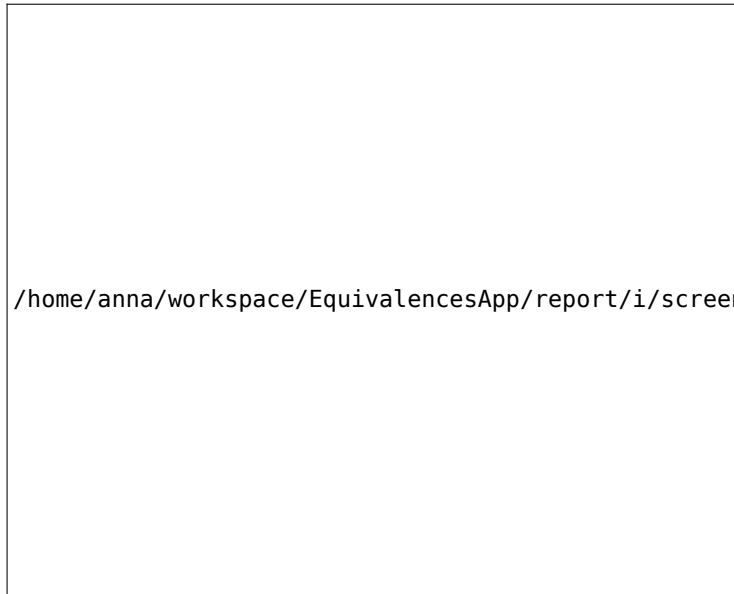Figure 8.5: Equivalences are manually entered without brackets.

Figure 8.6: Brackets have been implied and the tree is drawn accordingly.

8.6 Error Messages

A custom error message is displayed to the user under certain circumstances. A function `setErrorMessage()` is defined to set the error message to the passed in text and make the message visible.

No formulae have been entered

If the start button is clicked and both formulae have nothing in them then an error message saying *'Please enter two equivalent formulae'* is set (Figure 8.7).
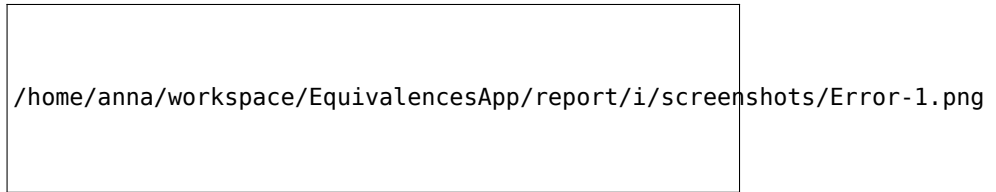
/home/anna/workspace/EquivalencesApp/report/i/screenshots/Error-1.png

Figure 8.7: Two formulae have not been entered

Incorrect Syntax

Subsection 3.2.3 describes how the parser handles syntax errors. When incorrect syntax is found it allows the incorrect formula to be parsed and sets a flag in the resulting tree. Before creating truth tables for each tree if either has its error flag set then the error message *'Incorrect syntax'* is set (Figure 8.8).
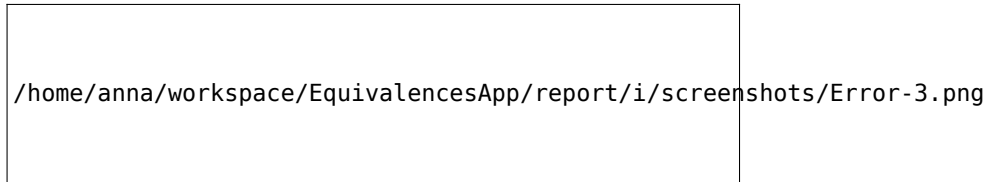

/home/anna/workspace/EquivalencesApp/report/i/screenshots/Error-3.png

Figure 8.8: Incorrect syntax has been entered

Formulae are not equivalent

If the test for equivalent truth tables fails then the intent is not created and the error message *'Not equivalent'* is set (Figure 8.9).
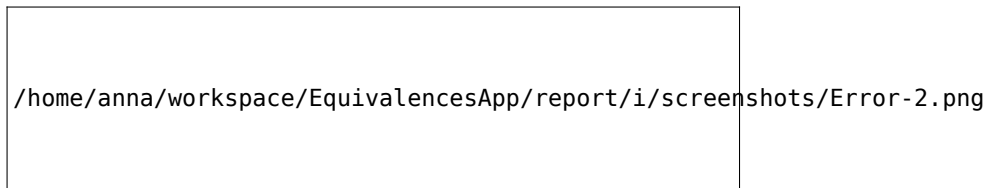

/home/anna/workspace/EquivalencesApp/report/i/screenshots/Error-2.png

Figure 8.9: The two formulae are not equivalent

8.7 Formation Tree Representation

8.7.1 The Formation Tree

Each logical connective in an formula can be written enclosed in parentheses. For example:

$$(a \rightarrow b) \wedge \neg a \text{ can be written as } ((a \rightarrow b) \wedge (\neg a))$$

Due to this property we can create a binary tree unique to each proposition (its formation tree). Each connective and propositional atom is represented by a node and a leaf respectively. This provides a clear and attractive way of displaying the formula but is too expensive for every day use. In our application we wish to display the formation tree of the current equivalences.

Displaying the formation trees allows a user to see an interactive representation of the equivalences where touching the nodes and leaves in the tree allows the user to apply rules to various parts of the tree.

8.7.2 The NP-Complete Problem

When planning to draw the formation trees to the app we assumed there would be a simple, classic algorithm for drawing neat, aesthetically pleasing trees. However, we discovered the problem was not that simple. Drawing an attractive Tree layout is an NP-complete[1] problem[16] and there are many tree-drawing algorithms attempting to solve the problem of drawing attractive trees. We reviewed many of these algorithms to find one suitable to use.

8.7.3 What Makes An Attractive Tree?

Although generating an attractive tree is a matter of taste, certain principles are widely agreed upon to be key to drawing an aesthetically pleasing tree. The first three are taken from Wetherell and Shannon's tree drawing algorithm for producing tidy drawings using the smallest width possible[13].

Aesthetic 1: Nodes at the same level of the tree should lie along a straight line, and the straight lines defining the levels should be parallel.

Aesthetic 2: A left son should be positioned to the left of its father and a right son to the right.

Aesthetic 3: A father should be centered over its sons.

---

[1]A problem is NP-complete if a given solution can be verified in polynomial time but there is no known efficient way to find a solution.

Although not mentioned in the original article, **Aesthetic 1** was also meant to guarantee the edges in the tree do not intersect except at the nodes. This was done by requiring that the relative order of nodes over any level be the same as the breadth-first traversal of the tree. Wetherall and Shannon's algorithm is fairly basic and has a deficiency that compromises the overall attractiveness of the resulting tree. It produces drawings that could be made narrower within the constraints of the aesthetics and are not entirely pleasing to the eye - nodes in certain subtrees are drawn too far apart due to the fact that their shape is influenced by the positioning of nodes outside that subtree. This leads to the asymmetry of the resulting tree. Therefore when Reingold and Tilford[14] set out to create a better tree drawing algorithm they introduced a fourth aesthetic:

Aesthetic 4: A tree and its mirror image should produce drawings that are reflections of one another; moreover, a subtree should be drawn the same way regardless of where it occurs in the tree.

Fulfilling this aesthetic requires us breaking Wetherell and Shannon's aim of minimum width so that all isomorphic subtrees (subtrees with a bijective/one-to-one mapping) are drawn the same. This is considered a more important principle than using the minimum width as our main aim is to create an attractive tree. This is an acceptable drawback as Supowit and Reingold proved that determining the minimum width under these aesthetics is NP-hard[15]. Therefore, we can use Reingold and Tilford's algorithm to draw our formation tree to produce an attractive result.

8.7.4 Reingold and Tilford's Algorithm

The proposed algorithm was based on the following heuristic:

"Two subtrees of a node should be formed independently, and then places as close together as possble"[13].

This is applied during a postorder traversal of the tree by taking the two subtrees of a node and calculating the contours (outlines) of those subtrees. The two subtrees are then positioned so they overlap at their roots before moving them apart until no two points are touching. This is done recursively at each level moving down the levels of the tree until the bottom of the shortest subtree is reached. Once the process is finished the position of the subtrees can be fixed relative to their parent node which is centered over them. Once the postorder traversal is complete a preorder traversal occurs to convert the relative positions of each subtree to their absolute coordinates in the completed tree.

8.7.5 Walker's Algorithm

Extending Reingold and Tilford's algorithm to to rooted ordered trees of unbounded degree as described in their paper produces layouts where some subtrees of the tree could have been dispersed better to create a more attractive layout. In 1990 Walker's

algorithm solved this problem by spacing out subtrees whenever possible, however in the initial algorithm presented the runtime was quadratic. Walker's algorithm was improved upon by Christoph Buchheim, Michael Jünger and Sebastian Leipert so it would run in linear time[18]. This is the algorithm we use in our application.

8.7.6 Abego TreeLayout

Abego TreeLayout is an external library which efficiently creates compact and highly cusomisable tree layouts. It is based on Walker's algorithm with the enhancements previously mentioned suggested by Buchheim, Jünger, and Leipert[19]. The software builds tree layouts in linear time so that even trees with many nodes are built quickly. TreeLayout separates the layout of the tree from the actual rendering, therefore it is suitable for drawing trees in an Android application because it is not limited to a specific output or format. Some problems were encountered in using this library, there are described in Subsection 7.3.2.

/home/anna/workspace/EquivalencesApp/report/i/abegolineartime.png

Figure 8.10: Illustrates the linear time behaviour of the Abego TreeLayout algorithm[20] and shows the applicability for large numbers of nodes. (Graph taken from the Abego TreeLayout documentation).

8.7.7 Drawing the Formation Tree

We created a custom view `DrawView` to draw the trees to the screen in the application. This is created by simply extending the Android `View` class and overriding the

`onDraw()` function. To add a custom view to our user interface we also need to define custom attributes in a resource element as shown in Listing 6.

```xml
<declare-styleable name='DrawView'>
    <attr name='showTree' format='boolean' />
</declare-styleable>
```

Listing 6: DrawView resource element

In `onDraw()` we determine whether we are drawing the top or bottom formation tree and call `setUpTreeLayout()` (shown in Listing 7) which uses the Abego TreeLayout to manage the layout.

```java
public void setUpTreeLayout(Location location) {
    DefaultConfiguration<Node> configuration = new
            DefaultConfiguration<Node>(gapBetweenLevels,
            gapBetweenNodes, location);

    FixedNodeExtentProvider<Node> nodeExtentProvider = new
            FixedNodeExtentProvider<>(20, 20);
    treeLayout = new TreeLayout<Node>(tree,
            nodeExtentProvider, configuration);


    ...
}
```

Listing 7: `setUpTreeLayout()` is called by `onDraw()` in `DrawView`

To use Abego TreeLayout we create a `TreeLayout` instance with our tree (accessible through the `TreeForTreeLayout` inferface), a `FixedNodeExtentProvider` (defines a fixed size for each node) and a `DefaultConfiguration` (configures the layout by defining parameters such as `gapBetweenNodes` and `gapBetweenLevels`).

`TreeLayout` calculates the position of each node so we can then paint the nodes and edges to the screen using Android Canvas.

8.7.8 Responding to Touch Events

By overriding `onTouchEvent()` in `DrawView` we can get the event action performed on the view. Using this we can check if the touched position is within the bounds of any of the nodes drawn to the screen. If the position is within the bounds of a node we display applicable rules for that node by calling `setRules()` on the current activity (`BeginEquivalenceActivity`). Selecting a node also forces a redraw so that we can highlight the selected node.

8.8 Rule Application

8.8.1 Rule Display

The rules are displayed in a `PopupMenu` on screen either above or below the selected tree. The rules are generated by the `RuleEngine` as described in Section 5.1.

Calling `setRules()` adds the generated rules iteratively to the `PopupMenu`. If the rule requires user input (eg. $\top := \bot \rightarrow a$) a SubMenu is created with the variables that can be applied as shown in Listing 8. Each entry in the `PopupMenu` contains the string representing the rule and the rule key. The key is required so we know which rule to apply on touch events.

```java
public void setRules(SparseArray<String> rules, Node
        selected, FormationTree selectedTree) {
    ...

    int key;
    for (int i = 0; i < rules.size(); ++i) {
        key = rules.keyAt(i);

        if (key < min_user_input_required) {
            rulesList.getMenu().add(Menu.NONE, key,
                    Menu.NONE, rules.get(key));
        } else {
            SubMenu sub = rulesList.getMenu().addSubMenu(
                    Menu.NONE, key, Menu.NONE, rules.get(key));
            SortedSet<String> vars = topTree.getVariables();
            vars.addAll(bottomTree.getVariables());

            for (String v : vars) {
                sub.add(Menu.NONE, key, Menu.NONE,
                        rules.get(key) + " using " + v);
            }
        }
    }
    rulesList.show();
}
```

Listing 8: Rules being added to the `rulesList` PopupMenu

8.8.2 Responding to PopupMenu Touch Events

Overriding `onMenuItemClick()` in `BeginEquivalenceActivity` allows us to determine which item was selected from the `PopupMenu` and and apply that rule to the

selected node. We can determine which rule should be applied by taking the key of selected item and applying that rule from our `BitSet` of rules using the `RuleEngine` (Section 5.1).

Applying a rule to the selected node adds a new `TextView` to the respective list of equivalences (either top or bottom). Once the rule has been applied the application will check for completion and cycles (Subsection 8.10.1). If either are found a `Toast` (a simple popup for displaying feedback) will be displayed alerting the user.

We also override `onDimiss()` so that when a user dismisses the `PopupMenu` without selecting a rule we can also redraw the formation tree to deselect the selected node.

8.9 Undo/Redo

A main feature of the application is intuitive undo and redo functionality. Selecting a previous equivalence in the list (from either end) should display the formation tree for that equivalence and grey out the undone equivalences below as shown in Figure 8.11.

8.9.1 Adding A New Formula

When a new formula is added it is either added to the top or bottom layout depending on where the call has come from. Throughout this section we will refer to the top layout, however the bottom layout functions in a similar way with layout inverted.

To add the new formula the top layout `addTextViewToTop()` is called. This sets up the text view by setting its layout parameters and customises how the view will look on screen. Its ID is set to the number of equivalences in the layout so far - which is used for ordering the equivalences when they are undone and redone.

Each text view has an `OnClickListener` set, the function of this will depend on where the text view is in the layout. If the text view displays the current formula then clicking it will simply hide or show the current formation tree. If the text view is a previous equivalence before the current formula then on click the application will undo up till this equivalence. If the text view is below the current formula then it has been previously undone and clicking it again will redo up to and including this equivalence.

After the `OnClickListener` is set up the text view will be added to the top layout, a line will be drawn below it and it will be pushed onto the top stack.

Figure 8.11: The user realises they have made a mistake and clicks the bottom formula on the left. This undoes the last two equivalences and sets them as undone in the center. Performing a new rule then removes the undone equivalences and adds the new rule to the bottom list.

8.9.2 OnClick Listeners

Undo

When a previous equivalence is clicked the application finds its ID (representing its position in the layout) and removes each equivalence from the current equivalence (found by the current size of the stack) until the clicked equivalence's position.

Each of these removed equivalences are then set up below the formation tree with another `OnClickListener` ready to be redone if required.

Redo

Clicking an undone equivalence will cause the application to loop through the undone equivalences from the first (least recent) up to and including the selected one. This removes them from this view and adds them back to the top layout above the formation tree.

Formation Tree

After either an undo or a redo the most recent equivalence in the layout should have its formation tree displayed. To make this easier we maintain a stack of trees. This contains only the trees of the equivalences in the current layout and not ones that have been undone. By maintaining this stack it allows the application to easily update the current formation tree by peeking into the stack after any undo or redo event and drawing the tree at the top of the stack.

8.10 Help

As this application is considered a teaching tool to help students learn about logic we want to provide them with some help when they make a mistake in their proof.

We considered the option of not allowing the user to make a mistake (for example by not being able to apply rules resulting in a cycle), however we decided this would allow the user to simply press buttons with no understanding of what they're doing and eventually reach a solution. This is not the aim because we want to help the user to improve. We agreed that the best approach to this is to allow them to make mistakes but alert them to those mistakes.

8.10.1 Detecting Cycles

Detecting when the user has completed a cycle is simple with our implementation because we maintain stacks of all the previous equivalences. Therefore, whenever a new rule is applied the application can loop through the stack and compare the new equivalence to all the previous equivalences. If any are the same then a cycle has been found and a `Toast` is displayed alerting the user to this fact.

We considered highlighting the equivalences in question to the user but decided this would make the process too easy for the user. The current equivalence will always be one end of the cycle so it is not too difficult for a user to look through their previous equivalences and spot the duplicate. By not highlighting the error immediately we allow the user to search through their previous equivalences and hopefully spot where they've gone wrong by themselves.

Once a cycle has been spotted by the user they can either choose to undo back to that point or simply undo the last turn and apply a different rule that will not result in a cycle if they believed they were on the right track. Detecting cycles in first order logic is slightly harder to handle because variables can be renamed, however we are only allowing the user to use each variable once (so there are no nested quantifiers with the same variable name) which helps solve this problem.

Evaluation

To evaluate the application we created a large set of tests to check the correctness of the logic in the application and we obtained feedback about the UI and features from the Undergraduate Project Fair.

9.1 Testing

Each section of the application needed to pass a selection of JUnit tests before development for the next section could begin. This is because the correctness of the logic in the application is essential to the success of the application. It also made development easier if we knew we would not encounter any serious errors from an already completed component.

9.1.1 Compiler

The compiler utilises the parser and the walker to convert a string input into its tree representation. All nodes in the internal tree structure should have correct indexes (key and depth values) and be located in the correct position in the tree.

We defined a string representation of the internal tree structure so we could compare these properties with an expected result. Each node in the tree is represented by its index followed by its value and any children as shown below:

$$a \wedge b \equiv 0 - 0 : \wedge(0 - 1 : a, 0 - 2 : b)$$

The index is displayed as a key-depth pair and any children are surrounded by brackets and separated with a comma.

Indexes

We defined a set of tests for each operator to ensure the indexes were being correctly labeled and placed correctly in the tree. Each operator has a test with the operator as

the root, the left child and the right child. For example, the 'and' operator has tests shown in Listing 9.

```java
@Test
public void testAnd() {
    FormationTree tree = compiler.compile("a^b");
    assertEquals("a^b: ", tree.toTreeString(),
            "0-0: ^ (0-1: a, 1-1: b)");
}

@Test
public void testAndLeft() {
    FormationTree tree = compiler.compile("(a^b)^c");
    assertEquals("a^b: ", tree.toTreeString(),
            "0-0: ^ (0-1: ^ (0-2: a, 1-2: b), 1-1: c)");
}

@Test
public void testAndRight() {
    FormationTree tree = compiler.compile("c^(a^b)");
    assertEquals("a^b: ", tree.toTreeString(),
            "0-0: ^ (0-1: c, 1-1: ^ (2-2: a, 3-2: b))");
}
```

Listing 9: Tests showing testing of internal tree structure for the 'and' operator

Order of Precedence

The order of precedence for operators is defined in the grammar file we pass in to *ANTLR*. We defined tests to check the parser parses the operators in the correct order. These tests are combinations of different operators written without brackets (unlike the tests defined above) to see what priority the parser gives to each operator. The test for the 'and' and 'implies' operators is shown in Listing 10.

```java
@Test
public void testAndImplies() {
    FormationTree tree = compiler.compile("a^b→a");
    assertEquals("a^b→a: ", tree.toTreeString(),
            "0-0: → (0-1: ^ (0-2: a, 1-2: b), 1-1: a)");
}
```

Listing 10: Testing order of precedence for 'and' and 'implies' operators

9.1.2 Rule Selection

It is important to ensure that only the correct applicable rules are displayed to the user when a node is selected as well as making sure that no applicable rules are missed. If an inapplicable rule was returned by the rule selector and a user selected it, not only would this cause the user confusion, it would throw an error and most likely crash the app.

Tests are defined to check the bitset returned by the rule selector is accurate. We defined a variety of tests so that each of the rules will be applicable in at least one test. In each test we create a bitset and set the expected values, then use the rule selector to return its bitset and compare the two bitsets for equivalence.

9.1.3 Rule Application

The rule applicator defines a series of methods used in applying rules to nodes. For example, applying commutativity has its own method so it can be used to apply commutativity to either 'and' or 'or' operators.

Simple Rules

For most rules we define two tests (some rules can use the same test because they use the same methods). The first tests the application of the rule on a simple formula and the second shows a more complex case. The commutativity example is shown in Listing 11.

```
// 0.  a^b |- b^a -- Commutativity
@Test
public void testAndCommutativity() {
    FormationTree tree = compiler.compile("q^p");
    ra.applyCommutativity((BinaryOperator) tree.findNode(0, 0));
    assertEquals("q^p: ", tree.toTreeString(),
            "0-0: ^ (0-1: p, 1-1: q)");
}

@Test
public void testAndCommutativityComplex() {
    FormationTree tree = compiler.compile("(¬q→r)^(pva)");
    ra.applyCommutativity((BinaryOperator) tree.findNode(0, 0));
    assertEquals("(¬q→r)^(pva): ", tree.toTreeString(),
            "0-0: ^ (0-1: v (0-2: p, 1-2: a), 1-1: →
            (2-2: ¬ (4-3: q), 3-2: r))");
}
```

Listing 11: Testing apply commutativity rule on simple and complex cases

Complex Rules

For rules where the selected node is being replaced by one of its children (such as absorption: $a \vee (a \wedge b) \equiv a$) we need to take extra care that the parent and children are set correctly. We define extra tests for these rules to test the methods are correct when they are applied to the children of a unary node and either side of a binary node. We use these to check the indexes are replaced correctly and the correct child replaces the parent node.

9.1.4 Truth Tables

Truth tables are used to confirm formulae are equivalent so that a user can begin a proof. If there are any errors in these tables or in how they are checked for equivalence a user could end up trying to prove that two nonequivalent formulae are equivalent.

Correct truth tables

For each of the operators used we tested that their resulting truth tables were as expected by asserting that they were equivalent to a manually entered array of values. This also ensured that the order of the rows in the tables were as predicted. Predictable row order is a key feature because it allows us to check tables for equivalence more easily.

Similar tests were also run using truth values ('$\top$', '$\bot$') instead of regular atoms to ensure they returned the correct truth values for themselves.

Testing equality

Tests were created for each defined rule to ensure that both side of the rule were equivalent. This not only tested that our testing for equivalence was correct but also that the rules we defined were accurate.

These tests checked all the cases described in Subsection 5.3.2 as they included tests for rules where each side has a differently sized table (such as the absorption rule).

We also defined some tests to ensure that checks for equivalence would fail in a variety of situations. If all the variables used in two propositional formulae are different they should fail the equivalence check. However, there are some edge cases which are exempt from this rule. If the formulae can be proved equivalent by our rules they should succeed.

An example of this case is $a \wedge \neg a$ and $b \wedge \neg b$. These formulae use totally different variables but both reduce to '$\bot$' so are solvable. The test for this case is defined in Listing 12.

```java
@Test
public void testDifferentVariables() {
    String s1 = "av¬a";
    String s2 = "bv¬b";
    FormationTree tree1 = compiler.compile(s1);
    FormationTree tree2 = compiler.compile(s2);
    TruthTable tt1 = new TruthTable(tree1);
    TruthTable tt2 = new TruthTable(tree2);
    assertTrue(tt1.testEquivalence(tt2));
}
```

Listing 12: Testing 'a∧¬a ≡ b∧¬b' edge case

9.1.5   Equivalence Generation

Testing equivalence generation is difficult due to its random factors. Instead, we have created tests to ensure that the contraints we set to define difficulty are always held. These constraints include how many variables will be found in a generated formula and the maximum depth of the resulting tree.

As the generation of formulae is random we set each test to run 100 times to ensure that the criteria is met every single time. We define three types of tests: the first tests generating variables and random binary and unary operators, the second tests the depth of the resulting tree, and the final type tests the proportion of truth values appearing in the resulting formula when a certain probability is set.

9.1.6   Test Results

All of these tests were run after each section was completed to ensure the correctness of the application remained intact as we extended the functionality. By implementing this wide range of tests (including edge cases) we know that the application is logically correct and is accurate in its application of rules and comparison of equivalences.

9.2   Undergraduate Project Fair

An undergraduate project fair was held so that 3rd and 4th year student could showcase their individual projects. I attended this to obtain feedback on the application, gain some insight into what features people found most useful and what could be improved. The fair was mainly attended by 3rd and 4th year students so it was a perfect opportunity to ask people who had already taken a logic course what they thought of the app. We tested our application on a Samsung Galaxy Nexus 7.

9.2.1 Feedback

The feedback from the fair was mostly positive and addressed a few key points: designing the application for Android; the UI design and propositional vs first order logic.

Using Android

The fact that the application was on an Android tablet was met favourably with most of the users agreeing that they would be more likely to use the application on a tablet in lectures than on a laptop. However, one student did raise the point that not all students have access to an Android tablet, whereas most have access to a laptop. This is a valid criticism of the application and if we were going to develop it further we would like to extend it to a web application to make it more accessible.

UI Design

We have aimed to create a functional logic application and it's fair to say that the UI has a very functional look to it. The design of the app received a mixed response and people had differing ideas about how it should look. It was generally agreed that the formation tree was easily readable and was aesthetically pleasing.

Some users suggested adding a 'pinch and zoom' gesture to allow easier movement around the tree. Others preferred the idea of a static layout believing that manipulating a tree too much would make it more difficult to compare with the other formation tree. We understand the need to be able to manipulate the tree in some capacity so that very wide trees can be displayed in their entirety.

Currently we simply allow the user to move the tree left and right using the buttons provided when the tree is too large for the screen. Although implementing a 'pinch and zoom' gesture to manipulate the tree would be more intuitive for some users we decided to stick with the simple approach.

Other comments included simple adjustments to the design such as centering the rule popup menu and providing some simple instructions on each screen to help the user know what to do at each stage. The idea to center the rule popup menu was discarded because it resulted in blocking much of the other formation tree which we believe is useful to see while applying rules. The other ideas were added into the design where possible and using them we believe the design is now more intuitive and easier to understand.

Propositional vs First Order Logic

At the time of the undergraduate fair the application only supported propositional logic. We wanted to incorporate first order logic into the application so decided to find

out which logic people would have found most useful when they were first learning logical equivalences.

Understandably, users agreed that first order logic was more difficult to learn. Therefore, a tool helping them understand first order would have been more appreciated and useful than an application only supporting propositional logic. For this reason we have completed support for first order logic and allow the user to choose which logic they want to solve an equivalence for on the first screen of the app.

Conclusion

Throughout the report we have demonstrated an Android application which can be used to prove logical equivalences. We have assessed each component in the project using a variety of tests to be certain of their correctness. In this chapter we discuss the initial objectives and assess how effectively they have been implemented as well as looking at any changes and improvements we made.

10.1   Achievements

ANTLR4

Using *ANTLR4* allowed the grammar used to be much more natural and easy to implement. *ANTLR4* is a very powerful tool which allowed us to generate the internal tree representation of a formula effectively.

Formation Tree

We wanted to display an attractive formation tree of the formulae in use by the application. We have shown that Walker's algorithm is extremely suitable for building aesthetically pleasing tree layouts. The trees are interactive to apply equivalence rules to the formula and we have proven these rules are correct through extensive testing.

To allow trees larger than the screen size to be displayed, buttons were added to allow simple navigation around the trees. This was a difficulty we had not anticipated at the beginning of the project, but the solution is effective.

Undo/Redo

The undo and redo functionality has been implemented as expected at the beginning of the project with very few minor changes to the design. We are able to undo and redo from each end independently as predicted.

Generated Equivalences

Extra features were added to the generation of equivalences due to problems we encountered during development. We focused on generating equivalences based on the user selected difficulty instead of a predicted difficulty (based off previous equivalences completed). The difficulty also now includes the ability to choose the probability of truth values ('$\top$' and '$\bot$') appearing in the generated equivalences as this proved to affect the difficulty of solving the equivalences substantially.

The resulting generated equivalences are widely varied and are representative of their difficulty. The application provides a user with a method of easily generating equivalences suited to their ability.

Help and Completion

The user receives a notification popup when a cycle is detected or the equivalence proof is complete. These features are not as comprehensive as hoped for at the beginning of the project. Initially, we wanted to give the user a breakdown of how successful their proof was by determining the optimal proof (using the fewest moves) for the equivalences. This proved to be a complex task and unfortunately could not be completed. However, by providing help (in the form of allowing them to select difficulties and detect cycles) to the user we allow them to minimise their proofs by learning and understanding the rules more intuitively through repeated use of the application.

First Order Logic

Our final extension was to support first order logic equivalences. We extended the *ANTLR* grammar and the custom walker to handle the extra symbols and added in extra rules to handle first order equivalences. A user can choose either propositional or first order logic and will be presented with an appropriate custom keyboard to enter those equivalences with. Equivalences can be generated for first order logic using the same parameters as for propositional. This is recommended over entering custom equivalences because we cannot check first order formulae for equivalence using truth tables. The equivalences formation trees are shown in a similar fashion, with the difference that the nodes can be different sizes due to quantifiers ($\forall x$) and predicate symbols ($Pxyz$) needing to be displayed in individual nodes.

10.2   Future Work

There is a large amount of scope for future improvements to our current implementation to make it more accessible and to improve the user learning experience.

10.2.1 Device Support

One complaint from user testers was that they did not own an Android tablet so would not have been able to use the app. A simple solution to this is obviously to increase the number of devices supported by the application.

As the underlying functionality is separated from the Android implementation it would be fairly straight forward to create a web application utilising these features.

We could also create a simple version of the app to use on Android smartphones that did not display the formation tree since this would consume far too much screen space and make the application unusable. Instead each individual element in the current formula would be clickable to bring up the list of applicable rules.

10.2.2 Custom Error Handler

In the future it would be useful to learn more about error handling in *ANTLR4* and to implement a custom error handler correctly. Although our workaround is functional and took significantly less time than creating a custom error handler. This would allow us to handle errors found by the parser as it walked through the formula instead of waiting until a tree is completely built to detect them.

10.2.3 More Complex Help

Increasing the help available to the user would be beneficial to new users struggling with the basic concepts of logical equivalences. More experienced users would have the option to turn help features off.

Calculating The Optimal Route

To help the user see where they could improve we would have liked to be able to display the shortest number of moves possible to complete the proof. We could compare this to the number of moves the user took and allow them to either try again or view an optimal solution.

Finding a general solution for generated equivalences would be fairly simple because they are generated by the repeated application of rules. Although this may not provide the optimal solution due to cycles and the possibility of a more direct route.

By repeated random application of rules the application could find a solution by brute force. For shorter equivalences this could be used to generate an optimal solution as there would not be too many permutations. However, as the equivalences got larger and the solutions required more rules, this may not be feasible and we would have to settle for just finding any solution and allowing the user to see if they could find a better one.

Suggested Next Move

When the user is unsure of what to do we could suggest the next move based on the solution calculated from the current equivalence. If an optimal solution had been generated from the initial formula and the user had detoured from the ideal route, the suggested move would be an undo back to when they were on track.

However this could result in users not learning anything and just using the application to solve their equivalences for them. To prevent this we could put a limit on the number of times this help feature could be used so that it was only used when they really needed it.

Future Cycle Detection

Using the repeated application of rules we could determine if a user was heading towards an inevitable cycle and warn them of this. The application could search a limited number of moves ahead and apply all the permutations of the rules in search of cycles. If every move resulted in a cycle the user has probably made a mistake and would be made aware of this.

10.2.4   Displaying Truth Tables

Since we create truth tables to evaluate propositional equivalences we could easily extend this to allow the user to see the truth table of a specific formula. We could add this as a separate section to either propositional or first order logic, allow a user to enter one or more formulae and display their truth tables. This would be a useful feature to help new users understand why certain equivalences were equivalent.

Bibliography

[1] Wikipedia, *Propositional Logic.* `http://en.wikipedia.org/wiki/Propositional_calculus`

[2] IEP, *Propositional Logic.* `http://www.iep.utm.edu/prop-log`

[3] Wikipedia, *Android (Operating System).* `http://en.wikipedia.org/wiki/Android_(operating_system)`

[4] Android Developer Guide, *OpenGL ES.* `http://developer.android.com/guide/topics/graphics/opengl.html`

[5] Natasha Lomas, *Android Still Growing Market Share By Winning First Time Smartphone Users.* `http://techcrunch.com/2014/05/06/android-still-growing-market-share-by-winning-first-time-smartphone-users/`

[6] Harry McCracken, *Who's Winning, iOS or Android?.* `http://techland.time.com/2013/04/16/ios-vs-android`

[7] Maarten Pennings, *Android development: Custom keyboard.* `http://www.fampennings.nl/maarten/android/09keyboard/index.htm`

[8] Colin Allen, Chris Menzel, *Logic Daemon.* `http://logic.tamu.edu`

[9] Imperial College London, *Pandora.* `http://www.doc.ic.ac.uk/pandora/newpandora/index.html`

[10] Enrico Franconi, *Propositional Logic Calculator.* `http://www.inf.unibz.it/~franconi/teaching/propcalc`

[11] BBC News, *BBC Google activations and downloads update May 2013.* `http://www.bbc.co.uk/news/technology-22542725`, May 15, 2013.

[12] Armour Technology Solutions, *5 Main Differences Between Web Apps and Native Apps.* `http://www.armourts.com/5-main-differences-between-web-apps-and-native-apps/`

[13] C. Wetherell and A. Shannon, *Tidier Drawings of Trees.* IEEE Trans. Software Eng., vol SE-5, pp. 514-520. 1979.

[14] Edward M. Reingold and John S. Tilford, *Tidier Drawings of Trees.* IEEE Transactions on Software Engineering, vol SE-7, no. 2, March 1981.

[15] Kenneth J. Supowit and Edward M. Reingold *The Complexity of Drawing Trees Nicely* vol 18, issue. 4, pp 377-392, January 1983.

[16] K. Marriott, *NP-Completeness of Minimal Width Unordered Tree Layout*, Journal of Graph Algorithms and Applications, vol. 8, no. 3, pp. 295-312 (2004). `http://www.emis.de/journals/JGAA/accepted/2004/MarriottStuckey2004.8.3.pdf`

[17] Bill Mill, *Drawing Presentable Trees.* `http://billmill.org/pymag-trees`

[18] Buchheim C, Jünger M, Leipert S. *Improving Walker's Algorithm to Run in Linear Time* Graph Drawing Lecture Notes in Computer Science Volume 2528, 2002, pp 344-353

[19] Buchheim C, Jünger M, Leipert S. *Drawing rooted trees in linear time.* Software—Practice and Experience 2006; 36(6):651–665

[20] *Abego TreeLayout* `https://code.google.com/p/treelayout/`

Rules List

```
Equivalences involving ∧
0.  a∧b          |-  b∧a
1.  a∧a          |-  a
2.  a∧⊤          |-  a
3.  ⊤∧a          |-  a
4.  ⊥∧a          |-  ⊥
5.  a∧⊥          |-  ⊥
6.  a∧¬a         |-  ⊥
7.  ¬a∧a         |-  ⊥
8.  a∧(b∧c)      |-  (a∧b)∧c     -- Associativity
9.  (a∧b)∧c      |-  a∧(b∧c)     -- Associativity
10. a∧¬b         |-  ¬(a→b)
11. (a→b)∧(b→a)  |-  a↔b
12. ¬a∧¬b        |-  ¬(a∨b)      -- De Morgan laws
13. a∧(b∨c)      |-  (a∧b)∨(a∧c) -- Distributitivity
14. (a∨b)∧c      |-  (a∧c)∨(b∧c) -- Distributitivity
15. (a∨b)∧(a∨c)  |-  a∨(b∧c)     -- Distributitivity
16. (a∨c)∧(b∨c)  |-  (a∧b)∨c     -- Distributitivity
17. a∧(a∨b)      |-  a           -- Absoption
18. (a∨b)∧a      |-  a           -- Absoption

Equivalences involving ∨
19. a∨b          |-  b∨a         -- Commutativity
20. a∨a          |-  a           -- Idempotence
21. ⊤∨a          |-  ⊤
22. a∨⊤          |-  ⊤
23. a∨¬a         |-  ⊤
24. ¬a∨a         |-  ⊤
25. a∨⊥          |-  a
26. ⊥∨a          |-  a
27. a∨(b∨c)      |-  (a∨b)∨c     -- Associativity
28. (a∨b)∨c      |-  a∨(b∨c)     -- Associativity
```

```
29. (a∧b)v(¬a∧¬b)  |-  a↔b
30. (a∧¬b)v(¬a∧b)  |-  ¬(a↔b)
31. ¬av¬b          |-  ¬(a∧b)        -- De Morgan laws
32. av(b∧c)        |-  (avb)∧(avc) -- Distributitivity
33. (a∧b)vc        |-  (avc)∧(bvc) -- Distributitivity
34. (a∧b)v(a∧c)    |-  a∧(bvc)      -- Distributitivity
35. (a∧c)v(b∧c)    |-  (avb)∧c      -- Distributitivity
36. av(a∧b)        |-  a            -- Absoption
37. (a∧b)va        |-  a            -- Absoption
38. ¬avb           |-  a→b
```

Equivalences involving ¬
```
39. ¬⊤     |-  ⊥
40. ¬⊥     |-  ⊤
41. ¬¬a    |-  a
42. ¬a     |-  a→⊥
43. ¬(a→b) |-  a∧¬b
44. ¬(av¬b)|-  a→b
45. ¬(a↔b) |-  a↔¬b
46. ¬(a↔b) |-  ¬a↔b
47. ¬(a↔b) |-  (a∧¬b)v(¬a∧b) -- Exclusive or of a and b
48. ¬(a∧b) |-  ¬av¬b     -- De Morgan laws
49. ¬(avb) |-  ¬a∧¬b     -- De Morgan laws
```

Equivalences involving →
```
50. a→a    |-  ⊤
51. ⊤→a    |-  a
52. a→⊤    |-  ⊤
53. ⊥→a    |-  ⊤
54. a→⊥    |-  ¬a
55. a→b    |-  ¬avb
56. a→b    |-  ¬(av¬b)
```

Equivalences involving ↔
```
57. a↔b    |-  (a→b)∧(b→a)
58. a↔b    |-  (a∧b)v(¬a∧¬b)
59. a↔¬b   |-  ¬(a↔b)
60. ¬a↔b   |-  ¬(a↔b)
```

Equivalences involving atoms
61. a |-   a∧a
62. a |-   a∧⊤
63. a |-   a∨a
64. a |-   a∨⊥
65. a |-   ¬¬a
66. a |-   ⊤→a
67. ⊥ |-   ¬⊤
68. ⊤ |-   ¬⊥

Equivalences involving user input
69. ⊥ |-   a∧⊥
70. ⊥ |-   a∧¬a
71. ⊤ |-   a∨⊤
72. ⊤ |-   a∨¬a
73. ⊤ |-   a→a
74. ⊤ |-   a→⊤
75. ⊤ |-   ⊥→a
76. a |-   a∨(a∧b)
77. a |-   a∧(a∨b)

First order - e and f represent any wff, t represents
              a wff with no free occurrences of x

Equivalences involving ∀
78. ∀x[¬e] |-   ¬∃x[e]
79. ∀x∀y[e] |-   ∀y∀x[e]
80. ∀x[e∧f] |-   ∀x[e]∧∀x[f]
81. ∀x[e∨t] |-   ∀x[e]∨t
82. ∀x[e→t] |-   ∃x[e]→t
83. ∀x[t→e] |-   t→∀x[e]
84: ∀x[t] |-   t

Equivalences involving ∃
86: ∃x[¬e] |-   ¬∀x[e]
87: ∃x∃y[e] |-   ∃y∃x[e]
88: ∃x[e∨f] |-   ∃x[e]∨∃x[f]
89: ∃x[e∧t] |-   ∃x[e]∧t
90: ∃x[e→t] |-   ∃x[e]→t
91: ∃x[t→e] |-   t→∃x[e]
92: ∃x[t] |-   t

Equivalences involving ¬
93: ¬∀x[e] |-   ∃x[¬e]
94: ¬∃x[e] |-   ∀x[¬e]

76

```
Equivalences involving ∧
95: ∀x[e]∧∀x[f]  |-  ∀x[e∧f]
96: ∃x[e]∧t      |-  ∃x[e∧t]

Equivalences involving ∨
97: ∀x[e]∨t      |-  ∀x[e∨t]
98: ∃x[e]∨∃x[f]  |-  ∃x[e∨f]

Equivalences involving →
99: ∃x[e]→t  |-  ∀x[e→t]
100: t→∀x[e] |-  ∀x[t→e]
101: ∃x[e]→t |-  ∃x[e→t]

Equivalences involving user input
102: ∀x[e] |-  ∀y[e{x->y}]
103: ∃x[e] |-  ∃y[e{x->y}]
104: t     |-  ∀x[t]
105: t     |-  ∃x[t]
```