Final Year Project Report

A Theorem Proving Assistant

Interim Report

Joe Duffin

A report submitted in part fulfilment of the
degree of BSc. (Hons.) in Computer Science

Supervisor: Henry Mcloughlin

School of Computer Science
University College Dublin

November 5, 2016

Contents

Project Specification

General Information

Calculational Theorem Proving is the name given to a particular style of mathematical proof which was developed during the 1980s by Feijen and Dijkstra. It emphasises the syntactic manipulation of expressions rather than manipulation based on an interpretation of the expression. In doing so it tries to let the notation do the work.

In order to do so the choice of notation and they layout of the proofs are important.

This style of proof has been taught in a number of our undergraduate modules over the last 10 years and is probably quite familiar to all of our undergraduates. An example proof would have been included here but the special symbols for the boolean operators were not available.

The aim of this project is to develop a tool which will assist the person proving the theorem by performing basic housekeeping tasks. So for example, the user should be able to select a sub-expression, select an appropriate rule to transform it, and have the system carry out the transformation and record the appropriate hint to show which rule was used and what the variable bindings were. It is important to mention that we are not interested in automatic theorem proving, the user will still be required to drive the proof but the system will perform the matching and applying of the rules.

It should be possible to store and retrieve partial or complete proofs. It should be possible to undo a series of steps and select different rules to apply. Once a theorem is proved it should be possible to add it to the rule base so it can be used in future proofs.

Mandatory Requirements

- The design of algorithms to a selected sub-expression with a subexpression of a rule.
- The design of an interface to allow users to select subexpressions and rules to apply.
- The design of appropriate algorithms to allow partial and complete proofs to be stored and retrieved.
- The design of appropriate algorithms to allow the user to undo steps in a proof.
- The implementation of the above functionality.

Discretionary Requirements

- Incorporating quantified expressions and their manipulation rules in the system.

Exceptional Requirements

- Incorporating additional small calculii such as Max/Min, Floor/Ceiling and GCD/LCM into the system.

Suggested Reading

- http://www.cs.utexas.edu/ EWD/
- http://www.mathmeth.com/
- Program Construction by Roland Backhouse
- A logical approach to discrete maths by David Gries

Abstract

Context

This report introduces the reader to predicate calculus and theorem proving in the style used by Dijkstra. It highlights the features of it that lend themselves to a computer application. Proving boolean theorems is a pattern matching excersise without need for interpretation of the expression at each stage of the proof. This report explores issues and solutions for building software package which facilitates theorem proving in a clear and easy manor.

Methodology

An exploratory approach was taken to not only resolving the major issues but finding them aswell. There is no road map to follow so a lot of time and resources were put into paths that led nowhere. In order to carry out this project I have had to define algorithms which are relied upon throughout, some of which have proven to be flawed and others invaluable. I shall discuss these.

Findings

At this stage of the project I believe that the major fundamental building blocks are in place. I have defined ways of representing expressions and carrying out steps of a proof. Simple pattern matching and replacement excersises can be conducted programatically. The algorithms used will be discussed indepth.

Introduction to Predicate Calculus

Predicate calculus works like this: given a set of axioms and theorems (rules) one can calculate, or derive, further theorems. Relying on those derived previously a full calculus can be evolved. This process relies heavily on pattern matching and this notion is vitaly important. With each step of a proof a hint is given which tells the reader which rule was used and what assignment was made to each variable in that rule. Dijykstra states that the notation used is vitally important for a clear and easy to read proof [**?**].

Prelimniaries

Before giving an example proof we must set some conventions for notation and give information about operators.

- Upper case letters such as X,Y and Z will be used as boolean identifiers.
- For this report I am only going to introduce 3 operators:

  Conjunction (boolean and): $\wedge$

  Disjunction (boolean or): $\vee$

  Equivalence (boolean equals): $\equiv$
- Equivalence holds the lowest precedence of the three operators and conjunction and disjunction share equal precedence meaning explicit bracketing must sometimes be used.
- Brackets can be introduced as long as precedence and operators operands are respected.
- Brackets can be removed from unary expressions.
- Rules will be numbered and named and be referenced to by their number or name.
- Each new line of a proof will be preceeded by an assignment and the corresponding rule. We call this "the hint".
  eg. $\{(X, Y := Z, Y).(5)\}$ reads as X and Y are assigned the values Z and Y respectively in rule 5.
- Axioms will be denoted with a $*$ to the left of it's number and theorems with a $\cdot$.

In order to give a sample proof I provide the following set of axioms and theorems.

$*$	(0) $[(X \equiv (Y \equiv Z)) \equiv ((X \equiv Y) \equiv Z)]$	$\equiv$ associative
$*$	(1) $[X \equiv Y \equiv Y \equiv X]$	$\equiv$ symmeetric
$*$	(2) $[X \equiv true \equiv X]$	$\equiv$ identity
$\cdot$	(3) $[X \equiv X]$	$\equiv$ reflexive
$\cdot$	(4) $[true]$	true
$*$	(5) $[X \vee Y \equiv Y \vee X]$	$\vee$ symmetric
$*$	(6) $[X \vee (Y \vee Z) \equiv (X \vee Y) \vee Z]$	$\vee$ associative
$*$	(7) $[X \vee X \equiv X]$	$\vee$ idempotent
$*$	(8) $[X \vee (Y \equiv Z) \equiv X \vee Y \equiv X \vee Z]$	$\vee / \equiv$

A Sample Proof

We will now attempt to prove that disjuntion distributes over its self.

$$
\begin{array}{ll}
 & (X \vee Y) \vee (X \vee Z) \\
= & \qquad \{(X, Y, Z := (X \vee Y), X, Z).(6)\} \\
 & ((X \vee Y) \vee X) \vee Z \\
= & \qquad \{(X, Y := (X \vee Y), X)).(5)\} \\
 & (X \vee (X \vee Y)) \vee Z \\
= & \qquad \{(X, Y, Z := (X, X, Y).(6)\} \\
 & ((X \vee X) \vee Y) \vee Z \\
= & \qquad \{(X := X).(7)\} \\
 & ((X) \vee Y) \vee Z \\
= & \qquad \{remove\ brackets\} \\
 & (X \vee Y) \vee Z \\
= & \qquad \{(X, Y, Z := (X, Y, Z).(6)\} \\
 & X \vee (Y \vee Z)
\end{array}
$$

This has yielded a new theorem so we shall record it so it is available for use in future proofs.

$$\cdot \ (9)\ [(X \vee Y) \vee (X \vee Z) \equiv X \vee (Y \vee Z)] \qquad \vee/\vee$$

Note that in the above proof no interpretation of any of the boolean expressions was done. It is purely a pattern matching exercise. At each step of the proof we used a single theorem to replace a section of an expression with an equivalent section. I plan to build a theorem proving assistant by exploiting this feature of the proof style.

Introduction to The Theorem Proving Assistant

What I intend to build

The theorem proving assistant will provide a user friendly software environment for assisting in proving predicate calculus theorems. It will not be an automated tool, but one which provides the user with a comprehensive and robust environment in which theorems can be proved. The software will take care of the "housework", such as automating the generation of the hint between each step.

The user will be presented with an environment in which they prove theorems. In order to permute an expression the following steps will be carried out by the user:

1. Select a theorem (or sub-expression of a theorem) to use as the current expression.
2. Select a sub-expression in the current expression (the software will ensure that only a valid subexpression can be chosen).
3. Choose a theorem to apply.
4. Make a selection of which replacement/assigment to use from that theorem.
5. View the new expression generated with a hint step describing which theorem and what assigment was used.
6. Continue to permute the new expression as the current expression until a conclusion has been drawn.
7. Save the proven theorem as a new theorem for future use.

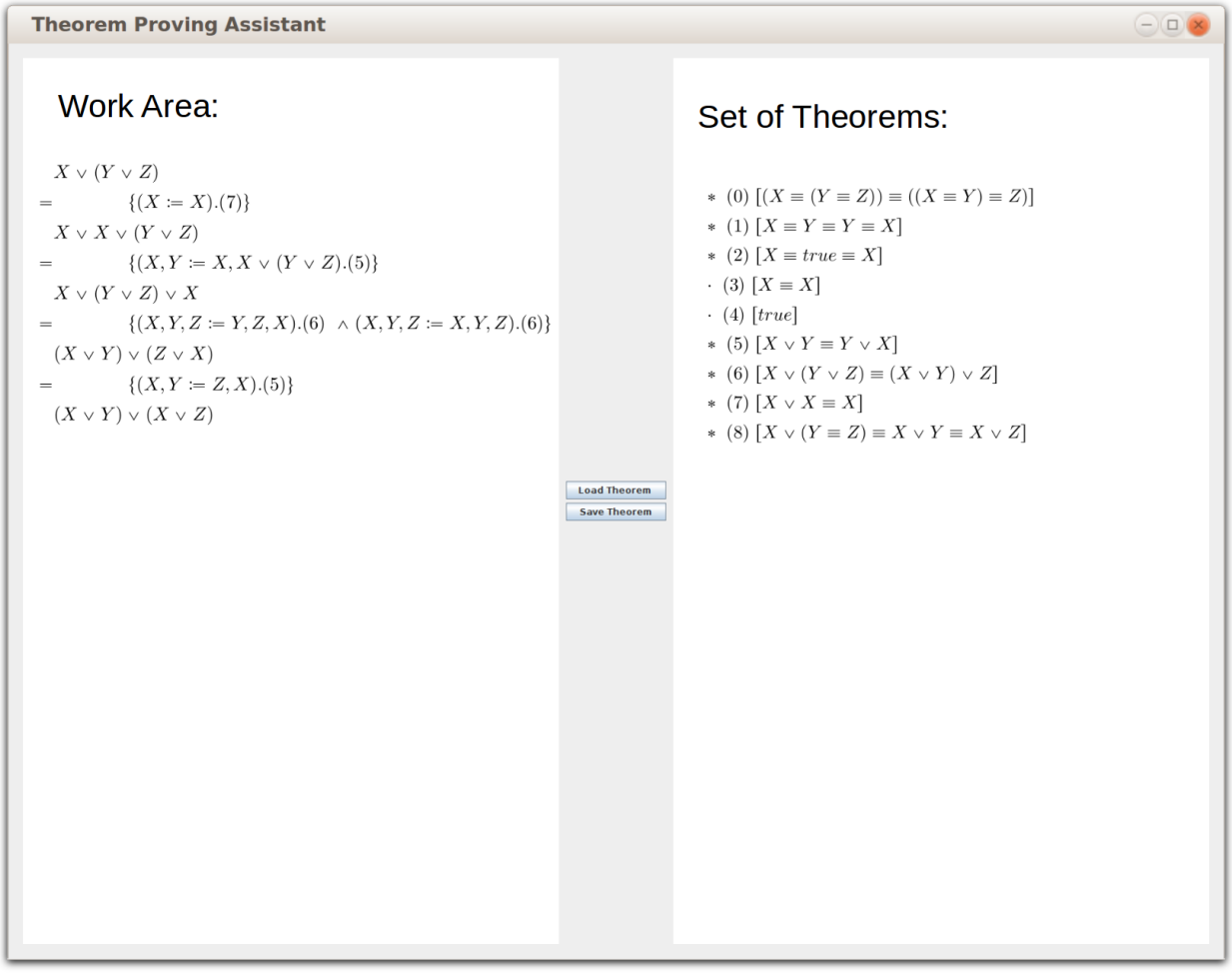
Further functionality will also provide the user with the ability to define their own workspaces where a parser will be used to transalate a psuedo language into a set boolean expressions to be used as theorems. This parser will translate input such as "X and Y" into the expression "$X \wedge Y$" in a format that the software can use.

What I hope it will look like

The interface will consist of two distinct areas. The left hand side is where the current expression and all steps taken to derive it will be displayed. It is here that a user will select a sub-expression of the current expression as described in step 2 above.

The righthand side of the interface will contain a list of numbered theorems. This list will be made up of loaded axioms as well as theorems that have been previously derived and saved by the user. It is here that a theorem will be chosen as described in step 3 above. The user will also have the ability to explore a theorem. This "exploration" will demonstrate from which theorem it was derived and which steps were taken in its derviation.

A button exists to move a theorem from the right hand side to the left hand side to start work and also a button to save the current expression as a theorem on the right hand side. The entire workspace state will be able to be saved at any moment and returned to for further work.



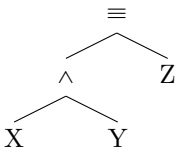
Explorations into the Field

Here I present work completed so far.

Expression Representation

When starting this project the only fact that was clear was the need to not represent expressions as strings. Immidiate issues associated with a string based solution are the lack of sense of precededence, the lack of pivots to easily commute around and no easy way to identify the begining and end of bracketed sections.

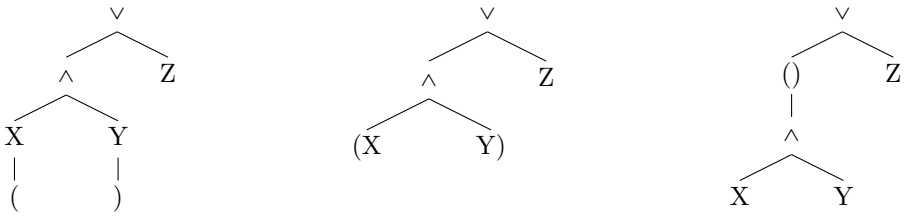
Syntax trees are a commonly used solution [**?**]. Leaves would be identifiers and nodes operators. Below I demonstrate an example with a simple boolean expression: $X \wedge Y \equiv Z$.



In order to yield the boolean expression from the tree we need only to perform an in-order travesal, printing the leaf or node's identifier or operator as we visit them. As a human we can read the expression from the tree by reading the identifiers and operators from left to right.

Depth is used to indicate precedence. One can see how precedence is maintained as the conjunction operator is at a deeper level than the equivalence. The two operators are commutative and swapping the left and right children of either node will yield an equivalent (in terms of ultimate value) tree.

Bracketing sections proved to be more of an issue. Below I demonstrate several options that were considerd with the boolean expresion $(X \wedge Y) \vee Z$.



The decision as to which option to use was made for me when I turned my attention the defining a valid substring.

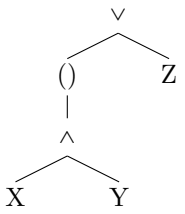
A Valid Sub-expression

A valid sub-expression is a fundamental part of the project. It is what a user will first select in order to carry out a replacement. For a sub-expression to be valid when selecting it we must respect precedence, bracketing and the number of operands associated with an operator. These requirements indicate that maybe a formal grammar will need to be defined but up until a parser is needed I wish to avoid this. I fear that having a grammar at this stage will steer me towards string based solutions. Initially I would like to stick to purely tree based solutions due to the issues associated with string manipulation.

To demonstrate valid and invalid sub-expressions of $X \wedge Y \equiv Z$ I present the table below.

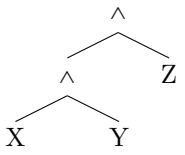
Valid	Invalid
$X \wedge Y$	$Y \equiv Z$
X	$\wedge Y$
Z	$\equiv Z$

Validating the selection of a sub-expression has to be a simple task. After drawing some syntax trees it became clear that any node's subtree is a valid sub-expression. For this reason the right most bracket option was chosen. Any node of that tree can be selected and its subtree is a valid sub-expression. We list the valid sub-exressions of $(X \wedge Y) \vee Z$ beside the tree and the corrolation becomes clear.



$(X \wedge Y) \vee Z$
$(X \wedge Y) \vee Z$
$(X \wedge Y)$
$X \wedge Y$
X
Y
Z

Initially this simple approach of using a node's subtree to define a sub-expression appeared comprehensive but when I went to extract all valid sub-expressions of an expression with associative operators I realised this was not possible. We use the expression $X \wedge Y \wedge Z$ to demonstrate this. Consider the following tree representation and list of valid sub-expressions as defined above.

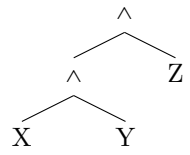


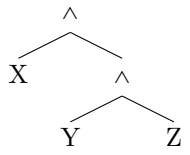
$X \wedge Y \wedge Z$
$X \wedge Y \wedge Z$
$X \wedge Y$
X
Y
Z

$Y \wedge Z$ is a valid sub-expression of $X \wedge Y \wedge Z$ but is not attainable from the above tree in its current state. We need to permute the syntax tree to "re-shuffle" it so $Y \wedge Z$ can exist as a subtree while still maintaining the value of the boolean expression.

Permutations of a Syntax Tree

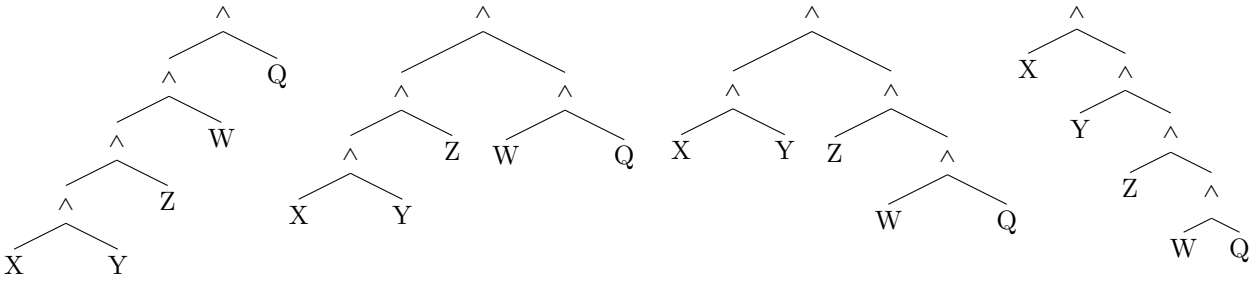
In order to achieve this re-shuffling of a tree we need to rely on a tree rotation algorithm called zigging [**?**]. Performing the zig operation on the deeper conjunction node will rotate the tree clockwise, causing the node to be the new root. X will be the left child of the new root with the previous root as its right child. The old roots right child, Z, persists, and the temporarily orphaned Y becomes Zs sibling. The following two trees present two permutations of a tree for $X \wedge Y \wedge Z$ attainable by zigging the left or right child of the root.



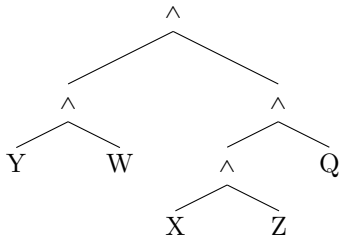


Now by unioning the the set of sub-expressions yielded from the two trees we have a complete set. This simple approach however does not scale. An algorithm combining commuting nodes, zigging nodes and zigging nodes from a depth of two is needed in order yield all valid sub-expressions of any tree. It is at this point that I must introduce the caveat that commutitivity is taken for granted. The algorithm will not only yield all sub-expressions permissisable by the associtivity of the operators, but also those permissable by the commutativity of the operators. Expressions such as $X \wedge Y$ and $Y \wedge X$ will be considered equivalent.

Consider the following series of tree representations of $X \wedge Y \wedge Z \wedge W \wedge Q$ and one can see how the whole tree is being "dragged" through the root node by performing the zig operation on the roots left child.



After each zig if we either do or do not commute the roots children and continue until no new trees are generated we will create every possible permutation. When the alorithm stops we have will yielded all 5! equivalent tree permutations. The following tree is one of them which will be produced at some point during permuting algorithm (I am not going to list all 5! of them!).



Recursively calling the described permute algorithm on the children of the root of each permutation will ultimately yield all valid sub-expressions. When attempting to identify valid sub-expressions of an expression all permutations of that expressions's tree must exist simulatiously; all possible tree arrangments for a given expression must be considered when validating or refuting an operation on an expression.

Matching, Mapping and Replacing

At this point I introduce a new theorem but no longer in terms of X and Y but P and Q.

$$\cdot \quad [P \wedge (P \vee Q) \equiv P] \qquad\qquad absorbtion0$$

We consider a state half way through a proof where the user's current expression is $X \wedge (X \vee Y) \equiv X \wedge Y$ and wishes to use absorbtion0 to replace the entire left hand side of the expression with X. The action is documented below with the notation we defined.

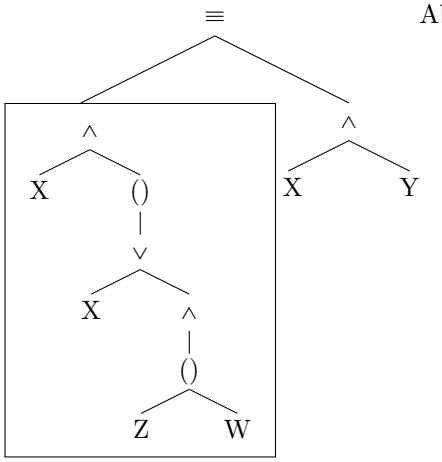
$$\begin{aligned} & X \wedge (X \vee Y) \equiv X \wedge Y \\ = & \qquad \{(P, Q := X, Y).(absorbtion0)\} \\ & X \equiv X \wedge Y \end{aligned}$$

As humans we can easily map P to X and Q to Y and do the replacement as it is in our nature to look for patterns and repetition. To design alorithms for a computer to do this is an complex task. To further complicate the matter we change the the boolean identifier Y on the left hand side of the expression to a boolean expression consisting of several operands. We redine the current expression and application of absorbtion0 as follows.

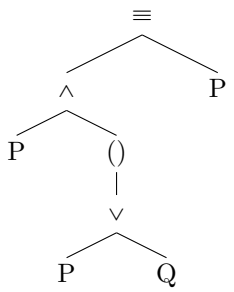
$$\begin{aligned} & X \wedge (X \vee (Z \wedge W)) \equiv X \wedge Y \\ = & \qquad \{(P, Q := X, (Z \wedge W)).(absorbtion0)\} \\ & X \equiv X \wedge Y \end{aligned}$$

In looking to match the users selection $(X \wedge (X \vee (Z \wedge W)))$ with a sub-expression of absorbtion0 we must compare it to every valid sub-expression of absorbtion0. Not only that, but we must handle the fact that the boolean identifiers in the rule and current expression are completely different. We start by drawing a tree representation of the current expression with the current expression highlighted and the rule, absorbtion0. Note that I have convieniently chosen to draw the correct permutation the rule. A software program will have to itterate through all permutations in attempts to find matches.

Current Expr:



Absorbtion0:



The matching algorthim first requests all valid sub-expressions of the rule with a root node that was a child of an equival node and has an operator which matches the root of the selection. In this case, all permutations of the subtree which starts at absorbtion0's single conjunction operator.

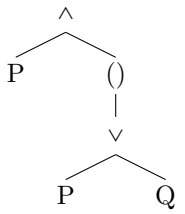
We walk each of these subtrees along side the current selection tree checking equivalence at each step. If a discrepency is found (such as operators that don't match) that subtree is discarded and we examine the

next. During the walk if an identifier is found in the rule's subtree, that identifier is added to a look-up table with the corresponding node from the selection. This lookup table will define the mapping used in the hint.

During the walk of the example trees provided there are three identifiers to be found in the subtree of absorbtion0.

1. On discovery of P (the left child of the subtrees root), it will map directly to the corresponding X in the current expresion. $P \to X$ will be added to the look up table.
2. On discovery of the second P we must be sure that this new mapping we are about to define corresponds with the mapping defined in step 1. If there is a discrepency this permutation of the subtree of the rule is not valid and thus discarded.
3. On discovery of the identifier Q we note that it does not correspond to another identifier, but a whole expression. That is not a problem and we add $Q \to (Z \wedge W)$ to the look-up table.

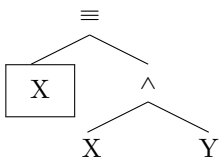
If the walk of any subtree of the rule that is checked completes, then we add that subtree and its look-up table to a list. This list will be the list of possible uses of a rule on a selection. Be aware that there may be multiple uses of a rule on a selection. In the example given the list will contain only one element, the following subtree and lookup table.



Look-Up Table
$P \to X$
$Q \to (Z \wedge W)$

To complete the replacement step there are a few small steps to take. None of which pose a large issue. We now need to remove the subtree from the permutation of the rule that contained it. In this case it will leave just the P node. Once we have the rule without the matched subtree we need to walk it and use the lookup table to replace its identifiers with their new nodes. If when walking we encounter an identifier not present in the look-up table we can use any identifier not present in the current expression.

In this example the rule without the matching subtree, P, will be walked and have its node replaced with its counterpart from the lookup table, X. At this point the matching and mapping is complete and we can return to the original expression we were working on and replace the current selection with the new tree (in this case, just an identifier node).



The replacement step has been performed, we know what rule we used and we also know what assignemnts were made into the rule. We can use this information to generate the needed hint and present the user with the outcome they expect. Here I re-express the replacement step from the start of this section to conclude the calculation of the step.

$$\begin{aligned} & X \wedge (X \vee (Z \wedge W)) \equiv X \wedge Y \\ = & \qquad \{(P, Q := X, (Z \wedge W)).(absorbtion0)\} \\ & X \equiv X \wedge Y \end{aligned}$$

Conclusion

Progress So Far

At this point of the project I have implemented (in Java) the algorithms described in this report. I have test cases written for each individual component aswell as test cases carrying out processes relying on several components such as finding all matches for a given sub-expression in a given rule. The test cases take no input as all trees are hard coded and they output to the console. All operations must be performed programatically.

It is not a requirement of this report to submit any implemented code but anything written so far is available for viewing on https://github.com/jj05y/Final-Year-Project-Theorem-Proving-Assitant. Note that this is a work in progress, attempts to download and run any code at this point will most likely end in heartache.

Next Steps

I next intend on focusing my efforts on two areas.

The GUI

The end result of this project will be a graphical user interface to allow a user to carry out the steps defined in this report. I need to research what is the best way to approach this. At the moment I am considering Java's Swing package or C# forms.

The GUI will need to add extra functionality seperate to the core of the theorem proving assistant such as load/save functionality, an undo stack and a smooth and clear interface.

The Grammar and The Parser

As well as a user having the ability to load a workspace with a set of hard-coded axioms to work with, I would like a user to be able to define their own workspace. For this a grammar and parser would be needed to translate an arbitrary input language into syntax trees. I am yet to give any thought to defining a grammar or the parser which will suit it.

References

[1] Carel S. Scholten and Edsger W. Dijkstra. *Predicate Calculus and Program Semantics*. 220 pages. ISBN: 978-1-4612-7924-2. Springer-Verlag New York, 1990.

[2] Jeffrey Ullman, Alfred Aho, and Ravi Sethi. *Compilers: Principles, Techniques, and Tools*. 796 pages. ISBN: 0201100886. Addison Wesley, 1986.

[3] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. 736 pages. ISBN: 978-0-470-39880-7. John Wiley & Sons Inc, 2010.