Final Year Project Report

A Theorem Proving Assistant

Joe Duffin

A thesis 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 sub-expression of a rule.
◦ The design of an interface to allow users to select sub-expressions 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 calculi 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 exercise without need for interpretation of the expression at each stage of the proof. This report explores issues and solutions for building a software package which facilitates theorem proving in a clear and easy manor. A software package written in Java is presented from both user experience and technical development points of view.

Methodology

There was 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 algorithms were defined which were relied upon throughout, some of which have proven to be flawed and others invaluable. These will be discussed. A modular approach was taken to the software development, focusing completing small tasks and writing relevant unit tests before moving on to the next.

Findings

Several major algorithms were defined to carry out the steps needed for theorem proving with the assistance of software. These algorithms were implemented in Java and a fully functioning software package was produced. It facilitates theorem proving over the predicate calculus with quantified expressions as well as other small calculi such as max/min and floor/ceiling.

Acknowledgments

So many people have had influential roles in the development described in this report as well as maintaining my sanity throughout. I must thank my supervisor, Henry McLoughlin for his advice and guidance. We discussed different approaches to solving problems at length. Some times we were in agreement, others not. This positive contention only sparked many trains of thought and it's a given that without Henry's influence this project would not have been as successful as it was.

Sean Russell stepped in as my supervisor during Henry's absence in semester two. Sean provided invaluable guidance in proof reading this report and in preparing my presentation. I feel a need to apologise to him for my atrocious spelling.

Others who have been forever giving with their time, listening to my mindless ramblings and aiding in solving the largest computer science problems I've ever faced would be my father Ray Duffin, my dear friend Sarah Doherty and my class mates Gary Mac Elhinney, Edwin Keville, Niamh Kavangagh and Nils Holgerson. Without these people keeping me going in the right direction I fear for the state that this project and my mental health would be in.

Introduction

It is in our human nature to look for connections and patterns. As soon as we (humans) see something we can associate with we latch onto it, we see where further exploration of it will take us. When we compare the capabilities of humans with that of a machine it is clear that machine wins in terms of raw processing power. When we attempt to compare this raw processing power to intelligence a strange divide occurs. It was only when an IBM computer (Deep Blue) was taught more patterns than a chess grand-master that it became ”smarter” [1].

> “Quite simply, humans are amazing pattern-recognition machines. They have the ability to recognize many different types of patterns - and then transform these ’recursive probabilistic fractals’ into concrete, actionable steps.”[2]

This report documents a software package called The Theorem Proving Assistant. Its key feature is pattern matching. It can be used to prove boolean and arithmetic theorems not by evaluating the value of the expression but by comparing the expression to known truths. Most importantly it matches the shape (or pattern) of an expression with known truths.

If I tell you that $x + y$ is that same as $y + x$ (or more formally $[x + y = y + x]$) then what can you tell me about $p + q$? It’s trivial to say that it is the same as $q + p$. However, attempting to define the simple logical steps one would need to instruct a computer perform this pattern matching operation is far from trivial. What about $w + (p \times q)$? Now it’s not so easy to state that it is the same as $(p \times q) + w$. At this point the problem has revealed it’s extra dimensions. How are were to write algorithms that can handle every case? Are we going to list every possible combination? That’s impossible. It’s in our human nature to solve these “recursive probabilistic fractals” easily. A computer on the other hand, is incapable without proper and clear instructions.

This report also documents algorithms devised and implemented to interpret expressions (which are fundamentally recursive probabilistic fractals) in such a way that pattern matching can be performed by a computer. These algorithms capture the essence of pattern matching as performed by a human to facilitate theorem proving in a fast, comprehensive and most importantly correct environment.

Introduction to Dijkstra's Theorem Proving Style

Predicate calculus theorem proving works like this: given a set of axioms and theorems (rules) further theorems can be calculated (derived). Relying on those derived previously a full calculus can be evolved. This process relies heavily on pattern matching and this notion is vitally 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 [3].

Prelimniaries

Before giving an example proof over the boolean domain 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 brief introduction I am only going to introduce 3 operators:

  Conjunction - boolean multiplication: $\wedge$ reads as "and".

  Disjunction - boolean addition: $\vee$ reads as"or".

  Equivalence - boolean equals: $\equiv$ reads as "equival".

– 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 operator's operands are respected.

– Brackets can be removed under the same conditions as their introduction.

– Rules will be numbered and sometimes named and can be referenced to by their number or name.

– Each new line of a proof will be preceded 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 disjunction distributes over itself.

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

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

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

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. The theorem proving assistant functions by exploiting this feature of the proof style.

A second proof is provided here for demonstration. It proves that $P \vee \mathit{true} \equiv \mathit{true}$.

$$
\begin{array}{ll}
 & P \vee \mathit{true} \\
\equiv & \qquad \{(X ::= P).(2)\} \\
 & P \vee (P \equiv P) \\
\equiv & \qquad \{(X, Y, Z := P, P, P).(6)\} \\
 & P \vee P \equiv P \vee P \\
\equiv & \qquad \{(X := P).(7)\} \\
 & P \equiv P \vee P \\
\equiv & \qquad \{(X := P).(7)\} \\
 & P \equiv P \\
\equiv & \qquad \{(X := P).(2)\} \\
 & \mathit{true}
\end{array}
$$

Again we record the proof:

· (10) $[P \vee \mathit{true} \equiv \mathit{true}]$ $\vee \mathit{zero}$

Further Notation and Calculi

The brief above introduction uses three operators, conjunction, disjunction and equivalence. Dijykstra's theorem proving style holds for many other operators and even other calculi. Those that are important for comprehension of this report and available for use in the Theorem Proving Assistant are described below.

Other Boolean Operators

Negation: $\neg$
Implication: $\Rightarrow$
Follows from: $\Leftarrow$
Not Equivalent: $\not\equiv$

Quantified Expressions

Quantified expressions are a way of gathering boolean expressions with the same operator, either conjunction or disjunction. They consist of 3 parts separated by colons.

1. The quantifier and dummy
2. The range
3. The term

The entire expression is wrapped in open-angled brackets as demonstrated in the examples below.

$\langle \forall i : r.i : f.i \vee X \rangle$ - For all i in the range r.i, f.i or X holds
$\langle \exists j :: g.j \rangle$ - There exists a j in all ranges where g.j holds

The Floor/Ceiling Calculus

The floor/ceiling calculi are defined over the real numbers. The floor function rounds a number down to the next nearest integer and the ceiling function rounds a number up to the next nearest integer.

- Floor of x: $\lfloor x \rfloor$
- Ceiling of x: $\lceil x \rceil$

The Max/Min Calculus

The max and min calculi are also defined over the real numbers. The max function yields the greater of it's two operands and the min function yields the lesser.

- Max of x and y: $x \uparrow y$
- Min of x and y: $x \downarrow y$

Lattice Theory

Lattice theory is defined over elements of some "type". Dijykstra gives some examples of familiar types to be either the natural numbers or even circles in a plane [4]. Lattice theory deals with a relation between two elements of a type. The relation is denoted with the symbol $\sqsubseteq$ and it is as a binary infix operator read as "under". It is demonstrated in the examples below.

$\langle \forall x :: x \sqsubseteq x \rangle$ - For any x it holds that x is under x
$\langle \forall x, y :: x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y \rangle$ - For any x and y it holds that x is under y and y is under x if x equals y

Introduction to The Theorem Proving Assistant

The Software

The theorem proving assistant is a user friendly software environment for assisting in proving theorems. It is 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 takes care of the "housework", such as automating the generation of the hint between each step.

The user is presented with an environment in which they prove theorems. To start a theorem the user needs an expression to work from. This is typed in in a plain text pseudo-language (see Appendix A) and after clicking start a parser will transform the input into the correct form. For example: "X and Y" will be transformed into the expression "$X \wedge Y$".

The expression which the user is working with at any one time will be refered to as the current expression.

In order to carry out steps of the proof the following steps are carried out by the user:

1. Select a sub-expression in the current expression (the software ensures that only a valid sub-expression can be chosen).
2. Choose a theorem to apply.
3. Make a selection of which replacement and assignment to use from a list of options.
4. View the new expression generated with a hint step describing which theorem and what assignment was used.
5. Continue to carry out steps with the new expression as the current expression until a conclusion has been drawn.
6. Save the proven theorem as a new theorem for future use.

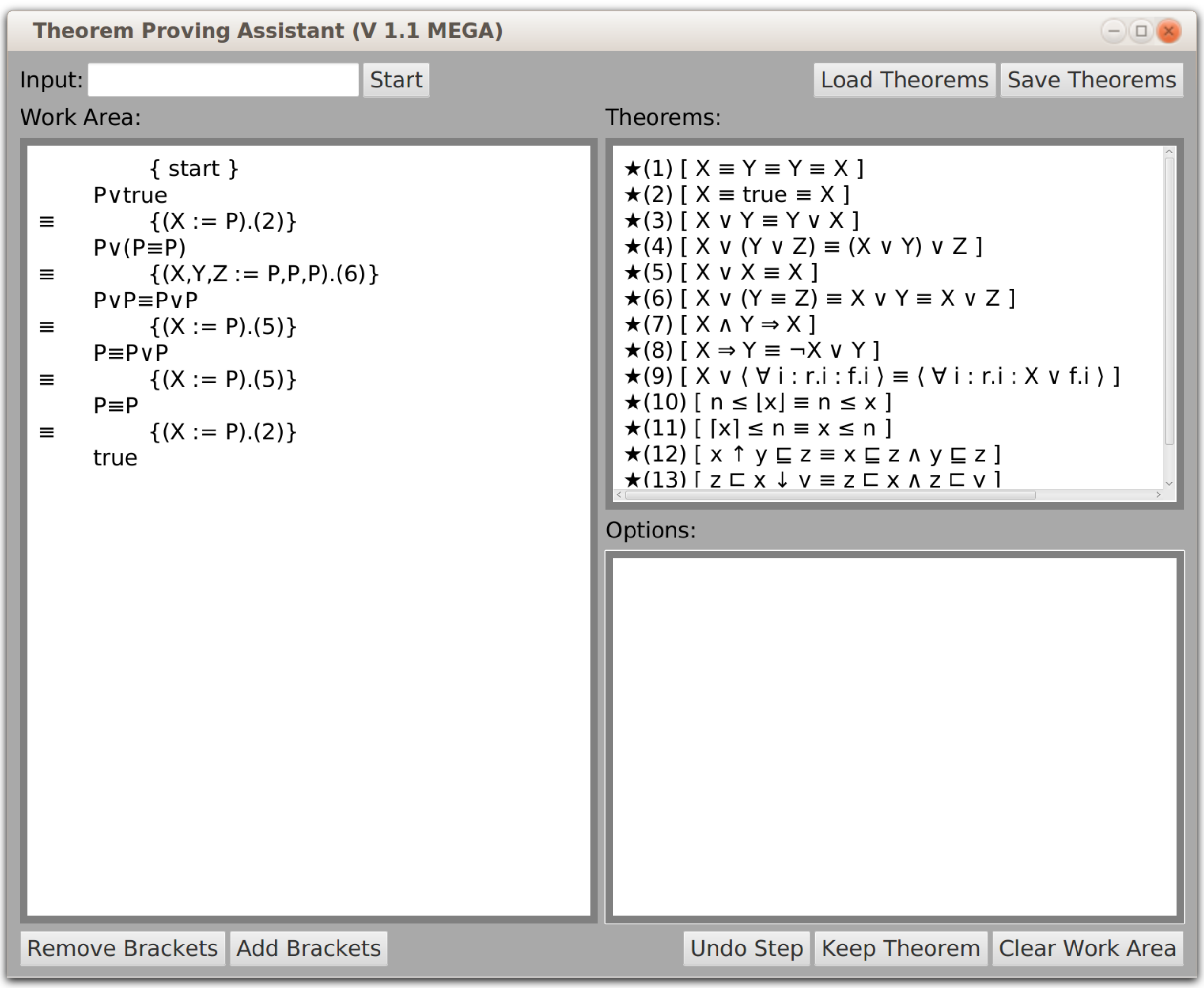
Further functionality allows the user to add and remove brackets where appropriate, undo a step, clear the work area and load or save theorem sets.

The Interface

The interface consists of three distinct areas. The left hand side is referred to as the work area. It is here where the current expression and all steps taken to derive it are displayed. It in this area that a user will select a sub-expression of the current expression as described in step 2 above.

The right hand side of the interface consists of both an area where a list of numbered theorems are displayed and also an area for replacement options to be displayed. The list of theorems is made up of loaded axioms as well as theorems that have been previously proved and saved by the user. It is here that a theorem is chosen as described in step 3 above and the replacement options generated are displayed in the options area. The user also has the ability to explore a theorem by clicking on it when the work area is clear. This ”exploration” shows the theorems derivation which documents the steps taken in order to prove it as well as giving the option to either delete a theorem or continue work on it.

The three areas are surrounded by several buttons as well as the input box for generating the starting step of a proof. The buttons are used to access the functionality of the theorem proving assistant. If the use of a button is not appropriate at a given time, that buttons functionality is disabled.



Functionality Overview

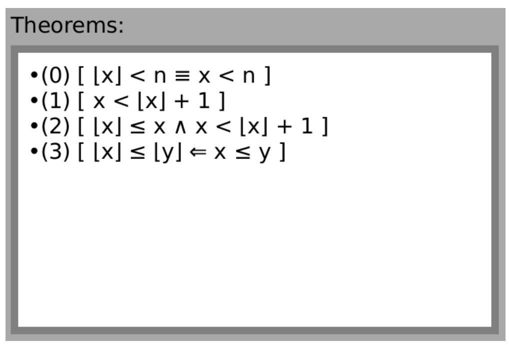
In this section the functionality of the Theorem Proving Assistant is explained. The steps are demonstrated with varying calculi to illustrate the versatility of the Theorem Proving Assistant.

Creating/Loading a Theorem Set

Theorem files are stored in a special format but can be added to or created by writing theorems in the plain text pseudo-language (see Appendix A). A $*$ or a $-$ is used to indicate whether this theorem should be interpreted as an axiom or not. Each theorem is placed on a separate line and the software will automatically handle their indexes. For example, if the user wants to use a selection of theorems relating to the Floor/Ceiling calculus the following would be added to a theorem file.

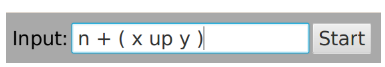
```
- |_ x _| < n == x < n
- x < |_ x _| + 1
- |_ x _| <= x and x < |_ x _| + 1
- |_ x _| <= |_ y _| <- x <= y
```

After clicking the Load Theorems button and selecting the file the theorems will be loaded into the theorems area.

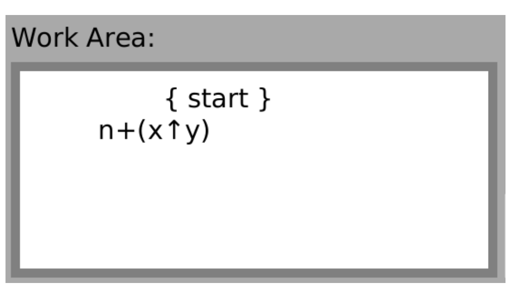


Starting a Proof

To start a proof the user types the starting expression into the input box in plain text. The following example is taken from the Max/Min calculus.



After clicking Start the input is parsed and presented to the user as the first proof step.

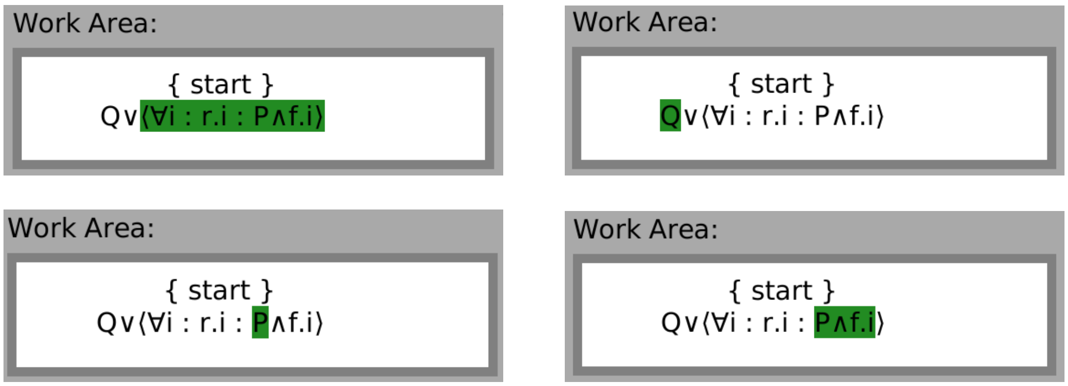


Performing a Proof Step

To perform a proof step there are three distinct operations: select a sub-expression, select a theorem and choose a replacement option. This example demonstrates permuting a boolean term within a quantified expression.

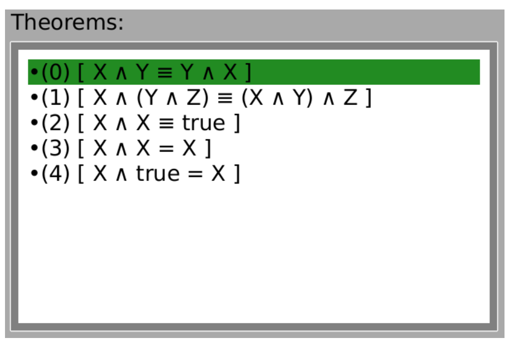
Select A Sub-expression

To select valid sub-expressions the user needs to select the operator around which the expression pivots. This is the lowest precedence operator in the required sub-expression. If multiple options are available, clicking multiple times will cycle the selection. Below shows the highlighting of several sub-expressions within the current expression.



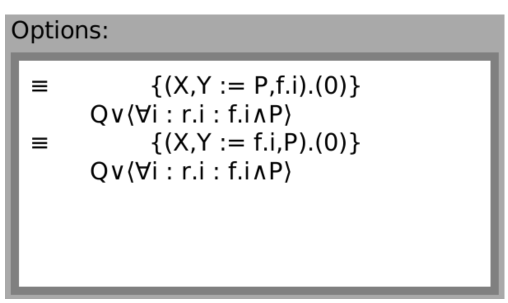
Select A Theorem

After choosing a sub-expression the user then needs to choose the theorem that they wish to apply. In this case we wish to commute P and $f.i$ around the $\wedge$ operator so we choose rule (0).

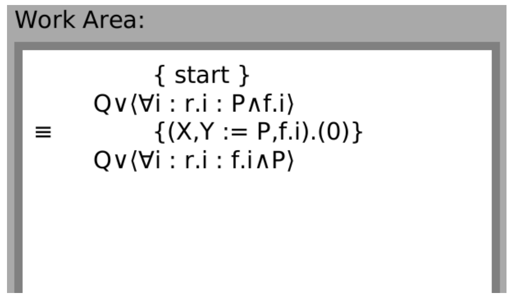


Select An Option

The list of options is automatically generated after clicking a theorem. Each option shows what the new current expression will be along with the hint which informs the user of the assignment and the rule used. In this case, after choosing theorem (0) only two options are available. They both yield the same end result but the assignment in the hint is different.

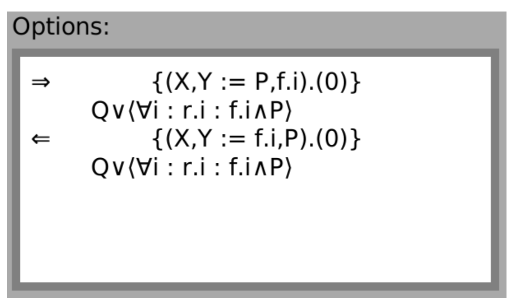


We choose the first option and the Work Area is updated with the new proof step. We are then free to select a sub-expression of the new current expression and continue generating new proof steps until a conclusion has been drawn.

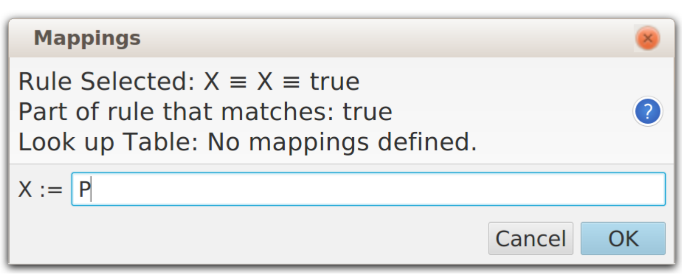


If the user is not satisfied with the options and clicks a different theorem, the software will generate a new list of options. This fast cycling and viewing of the options is the strongest point of the Theorem Proving Assistant. It allows easy and fast explorations of different paths during a proof.

There are two points to highlight about the generation of options. Firstly, note that the equivalence character is given with the hint, meaning that the current expression and the new generated expression are equivalent. This "transition" character is yielded from the rule. If the rule stated that $[X \land Y \Rightarrow Y \land X]$ then the transition character would be $\Rightarrow$. There is also a subtly taken into consideration with non-commutative transition characters, such as implication, where depending on the side of the rule which is used, the transition character must be pointing the correct direction. Below is the list of options generated from the given hypothetical theorem.



The second point to highlight is regarding unaccounted for Identifiers. Consider an example where the selection is *true* and the user wishes to use the rule $[X \equiv X \equiv true]$. It is clear that the replacement is $X \equiv X$ but the replacements $P \equiv P$ and $Q \equiv Q$ are equally valid. Should this case arise the user is prompted with a dialogue to enter the desired mapping. This new mapping is to be input in plain text.



Keeping a Theorem

When a conclusion has been drawn such as the proof of Absorption.0 $[X \wedge (X \vee Y) \equiv X]$ the user can save this theorem for future use. The Work Area below shows the entire proof.

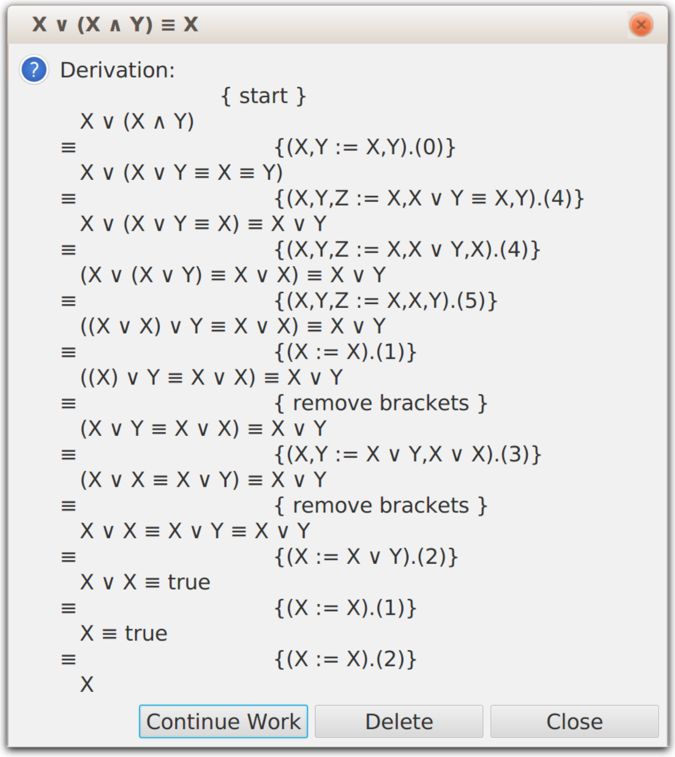
```
Work Area:
        { start }
    Xv(X∧Y)
≡       {(X,Y := X,Y).(0)}
    Xv(XvY≡X≡Y)
≡       {(X,Y,Z := X,X v Y ≡ X,Y).(4)}
    Xv(XvY≡X)≡XvY
≡       {(X,Y,Z := X,X v Y,X).(4)}
    (Xv(XvY)≡XvX)≡XvY
≡       {(X,Y,Z := X,X,Y).(5)}
    ((XvX)vY≡XvX)≡XvY
≡       {(X := X).(1)}
    ((X)vY≡XvX)≡XvY
≡       { remove brackets }
    (XvY≡XvX)≡XvY
≡       {(X,Y := X v Y,X v X).(3)}
    (XvX≡XvY)≡XvY
≡       { remove brackets }
    XvX≡XvY≡XvY
≡       {(X := X v Y).(2)}
    XvX≡true
≡       {(X := X).(1)}
    X≡true
≡       {(X := X).(2)}
    X
```

After clicking the Keeper button the theorem is added to the list of theorems. Note that the equivalence operator ($\equiv$) is used to join the left and right sides of the theorem. If an operator of higher precedence was used within the proof as a transition character, such as implication, then that operator would be used to join the two sides.

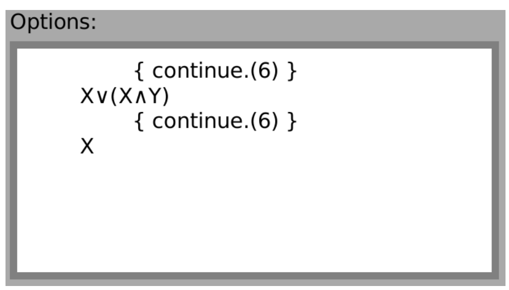
```
Theorems:
★(0) [ X v Y ≡ X ∧ Y ≡ X ≡ Y ]
★(1) [ X v X ≡ X ]
★(2) [ X ≡ X ≡ true ]
★(3) [ X ≡ Y ≡ Y ≡ X ]
★(4) [ X v (Y ≡ Z) ≡ X v Y ≡ X v Z ]
★(5) [ X v (Y v Z) ≡ (X v Y) v Z ]
•(6) [X v (X ∧ Y) ≡ X]
```

The new theorem has been added to the list of theorems and marked with a dot. This indicates that this theorem was proved and was not a given axiom.

This kept theorem can be clicked on to be explored. (This feature is only available when the work area is clear). This exploration shows all the proof steps that were taken when proving the theorem.



There is also the option to delete the theorem or to continue work. If Continue is chosen a list of continuation proof steps are given as options. These options are the theorem split up on the lowest precedence operator in the theorem.



Saving all Theorems

If the user chooses to save the theorem set the theorems are converted to a special format and stored as plain text. This format contains the theorem as well as information about it's derivation and whether or not it is an axiom. The saved theorem file can be extended with plain text theorems in the same format described above in "Creating/Loading a Theorem Set".

Backend Algorithms - The Engine

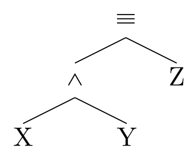
The software appears to be string manipulation based but the engine driving the system only generates these strings at the moment they are needed. The backend engine has most of the responsibility of allowing the user to make selections, match them in rules and perform replacements.

Here the fundamental methodologies and algorithms of this engine are discussed.

Expression Representation

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

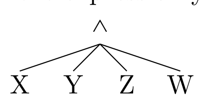
Syntax trees are a commonly used solution [5]. Leaves are identifiers and nodes operators. Below is presented an example with a simple boolean expression: $X \wedge Y \equiv Z$.

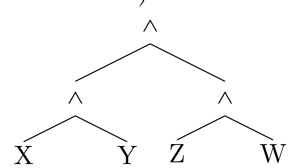


In order to yield the boolean expression from the tree we need only to perform an in-order traversal, printing the leaf or node's identifier or operator as we visit them. As a human we can read the expression from the tree by reading the identifiers and operators from left to right.

Depth is used to indicate precedence. It is evident 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.

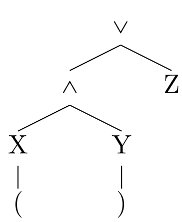
The option existed to use n-ary trees for expressions such as $X \wedge Y \wedge Z \wedge W$. Below two trees are depicted for that very expression. The tree on the left is an n-ary tree and on the right, an equivalent binary tree (equivalent in terms of the expression yielded from an in-order traversal).

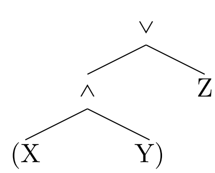


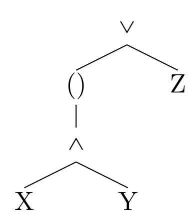


While n-ary trees appear to be an attractive option with small condensed trees they were disgarded as it was felt it is important to stay true to the operator. Conjunction is a binary operator so it should be represented with a binary node. Doing otherwise would be an implicit attempt at redefining conjunction to be an n-ary operator.

Bracketing sections proved to be more of an issue. Below several options are demonstrated that were considered with the boolean expression $(X \wedge Y) \vee Z$.







The decision as to which option to use was implicitly made when attempting to define a valid sub-expression.

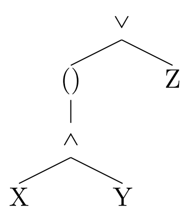
A Valid Sub-expression

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 with a corresponding parser but this was avoided as having the ability to parse strings lends itself to string based solutions. The engine needs to work with purely tree based solutions due to the previously discussed issues associated with string manipulation.

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

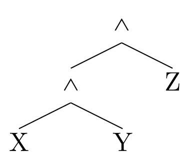
Valid	Invalid
$X \wedge Y$	$Y \equiv Z$
X	$\wedge Y$
Z	$\equiv Z$

Validating the selection of a sub-expression has to be a simple task. Examining drawn syntax trees makes it clear that any node's subtree is a valid sub-expression. For this reason the right most bracket option is used throughout. Any node of that tree can be selected and its subtree is a valid sub-expression. We list the valid sub-expressions of $(X \wedge Y) \vee Z$ beside the tree and the correlation 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 attempting to extract all valid sub-expressions of an expression with associative operators it became clear that 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.

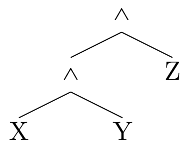


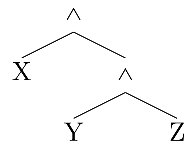
$X \wedge Y \wedge Z$
$X \wedge Y \wedge Z$
$X \wedge Y$
X
Y
Z

$Y \wedge Z$ 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 "reshuffle" it so $Y \wedge Z$ can exist as a subtree while still maintaining the value of the boolean expression.

Permutations of a Syntax Tree

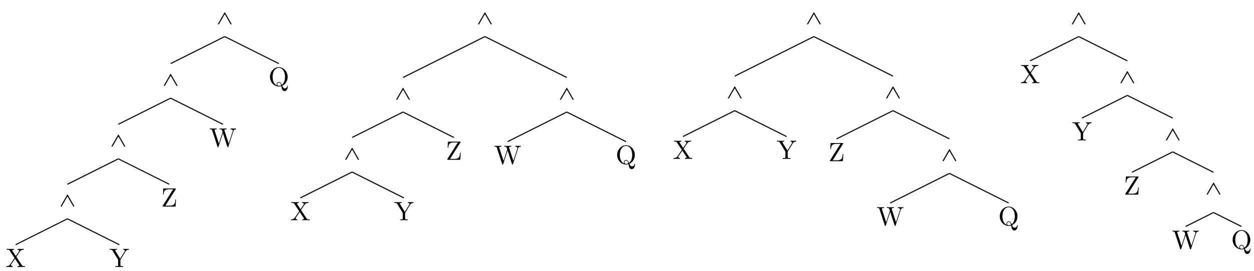
In order to achieve this reshuffling of a tree we need to rely on a tree rotation algorithm called zigging [6]. Performing the zig operation on the deeper conjunction node will rotate the tree clockwise, causing the node to be the new root. X will be the left child of the new root with the previous root 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.



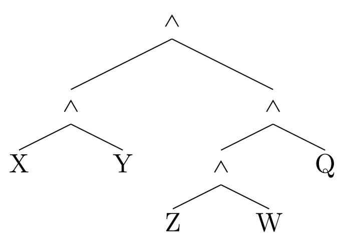


Now by taking the union of the sets of sub-expressions yielded from the two trees we have a complete set. This simple approach however does not scale. An algorithm combining zigging nodes and zigging nodes from a depth of two is needed in order yield all valid sub-expressions of any tree.

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 we add a copy of the tree as it stands to a set and continue until no new trees are generated. This we will ultimately create every possible permutation. When the algorithm stops we have will yielded all equivalent tree permutations. The following tree is one of them which will be produced at some point during permuting algorithm.



When attempting to identify valid sub-expressions of an expression all permutations of that expression's tree must exist simultaneously. All possible tree arrangements for a given expression must be considered when validating or refuting an operation on an expression.

Matching, Mapping and Replacing

At this point a new theorem must be introduced 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 absorption0 to replace the entire left hand side of the expression with X. The action is documented below with the notation we defined.

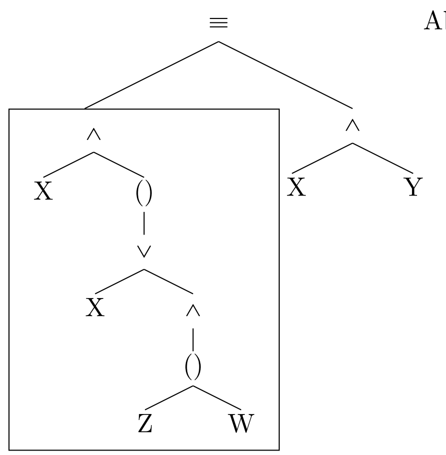
$$\begin{aligned} & X \wedge (X \vee Y) \equiv X \wedge Y \\ = & \qquad \{(P, Q := X, Y).(absorbtion0)\} \\ & X \equiv X \wedge Y \end{aligned}$$

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 these patterns. To design algorithms for a computer to do this is a 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 redefine the current expression and application of absorption0 as follows.

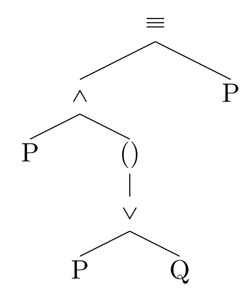
$$\begin{aligned} & X \wedge (X \vee (Z \wedge W)) \equiv X \wedge Y \\ = & \qquad \{(P, Q := X, (Z \wedge W)).(absorbtion0)\} \\ & X \equiv X \wedge Y \end{aligned}$$

In looking to match the users selection $(X \wedge (X \vee (Z \wedge W)))$ with a sub-expression of absorption0 we must compare it to every valid sub-expression of absorption0. 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 conveniently chosen to draw the correct permutation the rule. The software program actually iterates through all permutations in attempts to find matches.

Current Expr:



Absorbtion0:



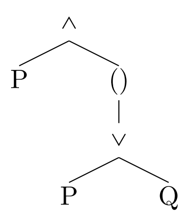
The matching algorithm first requests all valid trees representing the rule with a root node that was a child of an equival node and has an operator which matches the root of the selection. In this case, the single permutation of the subtree which starts at absorption0's single conjunction operator.

We walk each of the yielded subtrees along side the current selection tree checking equivalence at each step. If a discrepancy 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 absorption0.

1. On discovery of P (the left child of the subtrees root), it will map directly to the corresponding X in the current expression. $P \to X$ 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 discrepancy 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.

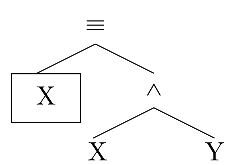
If the walk of any subtree of the rule that is checked completes, then we add that subtree and its look-up table to a list. This list will be the list of possible uses of a rule on a selection. Be aware that there may be multiple uses of a rule on a selection. In the example given the list will contain only one element, the following subtree and lookup table. One final check needs to be carried out to determine if there is a variable in the rule which has no baring on the selected sub expression, ie: it has no mapping automatically defined. In this case the mapping is open, and any value can be chosen.



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.

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 assignments 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 the replacement step from the start of this section is presented again to conclude the calculation of the step.

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

Frontend Algorithms - The User Interface

Two major algorithms were needed to link the backend engine with the graphical user interface. Before introducing them the term "bit" must be introduced. In this context a bit refers to a character displayed to the user. A bit has many subtle properties which allow the software to perform. A further extension of this is the notion of a Bit Box. The software is implemented with JavaFX, and the container used to house an entire expression is a HBox (horizontal box) so they are referred to as Bit Boxes.

The Bit Box Maker

To give the illusion of a string based piece of software a Bit Box is used to house an expression. For expressions with no bracket-type (brackets or floor/ceiling) or unary operators, a simple in-order traversal of any one of the trees for an expression will suffice. A bit is created when visiting each node during an in-order traversal, the bits character corresponds to the visited nodes character and that new bit is added to the Bit Box.

If a bracket-type node is found during the traversal then two bits need to be created, the opening and closing bits. In this case the open bit inserted before continuing the traverse. Only when the recursive tree walker returns to to the level of this bracket-type node is the closing bit inserted. It can be thought of a special case "pre & post" order traversal.

In the case of a unary operator such as negation the node is simply visited, and a bit added to the Bit Box, before continuing to traverse. It is treated as a pre-order visit. Both of these exceptions are special cases that are handled by the tree walker.

The Associator

It is the role of the Associator to associate bits and their corresponding nodes in the syntax trees. It is not a bijective mapping as one bit can map to many nodes in equivalent (in terms of the expression they represent) syntax trees. Each bit has a list of nodes with which it is associated. Should the bit belong to a bracket-type node then it has a "bracket-buddy". This is a reference to the corresponding closing (or opening) bracket. Only one of these two will hold the list of bracket-type nodes.

Each equivalent syntax tree is walked and the bits iterated across. An in-order traversal of the tree will yield the nodes in the same order as they appear in the Bit Box with a couple of exceptions. Bracket-type nodes and Unary nodes have to be handled carefully as they break the ordering of the in-order traversal. Instead of using convoluted logic to track the locations of these bits they are simply pushed onto a stack. Similarly, any opening bracket-type bit is pushed onto a parallel stack. Closing bracket-type bits are skipped as they are not needed, the bracket-buddy reference is used instead. After completion of the walk of each tree and single iteration of the bits, (which conclude at the same time), we have two sets of stacks. One for bracket-type operators with a bit stack and a node stack, and another for unary operators with same. The bit and Node stacks are popped in tandem and associated as their order in the stacks correspond.

The nodes and bits are associated in this way so that when a user clicks on any bit, it is known what valid sub expressions that bit is a root of as it has a reference to that subtrees root. We need only walk that subtree and highlight each bit associated with the visited nodes in order to highlight, and thus select, a valid sub-expression. Repeatedly clicking on the same bit will cycle the subtrees with which is it associated and highlight different sub-expressions.

The Parser and Grammar

In order to start theorems the user needs to type in a starting expression. For simplicity, A plain text pseudo-language was defined (see Appendix A) with which a user can specify the starting expression. A lexer uses a string tokeniser to iterate through the input symbols and then a recursive descent parser (RDP) builds the abstract syntax tree for the expression (assuming it's valid).

The EBNF Grammar

In order to build an RDP a grammar was needed. JJTree and JavaCC were considered as a tools to build an RDP but ultimately it was deemed too heavy weight. A grammar was defined in extended BackusNaur form (EBNF) (see Appendix B). The EBNF grammar consists of terminal symbols and non-terminal production rules. The terminal symbols define legal characters and strings in the expressions and the non-terminal production rules define the way in which the symbols can be legally arranged in order to create expressions.

- Terminal symbols are enclosed in angle brackets (<>) and have a sequences of characters associated with them. Eg: <ID>:= [A-Z,a-z]+
- Production rules have a left hand side and a right hand side. The left hand side is a non-terminal and the right hand side is a combination of terminals and non-terminals.
  Eg: Max := $Val\{\uparrow Val\}$ (where val produces a terminal)
  The { and } brackets indicate an optional amount of the terminals and non terminals within it. This allows for arbitrarily long expressions with the Max operator.
  Alternate production rules are indicated with a |.

The entire grammar is documented in Appendix B.

The Parser

The parser has two key roles; to build the syntax tree for the expression and to respect the laws of precedence of the operators while doing so. To build the parser a very simple approach was followed where a method was defined for each non-terminal production. The order in which these methods are called is the order of precedence of the operators. Each method follows the same framework except for the final method, factor(). The factor() method has special code to define the different types of node to create, depending on the value yielded by the lexer.
Below is one of the standard methods. Seeing as or() is called from within implication() it will cause disjunction expressions to be created at a deeper level in the tree, and as a result the disjunction operator will be considered to have a higher precedence than that of implication.

```
public void implication() {
    or();
    while (symbol == Lexer.IMPLICATION) {
        INode n = new BinaryOperator(Operators.IMPLICATION, null, null);
        n.children()[0] = root;
        or();
        n.children()[1] = root;
        root = n;
    }
}
```

Using this RDP will ensure that any legally constructed expressions, written in plain text, will yield a syntax tree which respects the precedence of the operators. This is essential for the algorithms in the engine of the Theorem Proving Assistant to work.

Design Details and Implementation

Tools Used

Java

The Theorem proving assistant is almost exclusively written in Java. It must be compiled and executed with Java 1.8 or greater. Java was chosen for several reasons.

- High portability of written applications.
- Multiple inheritance allows nodes to be given characteristics. When walking trees it is essential to be able to identify the abilities of a node. For example, all of the terminal classes (ArrayAndIndex, Identifier, Literal and QuantifiedExpression) implement INode and ITerminal.
- A wide variety well documented of libraries exists such as JavaFX and GSON.
- Memory management is automatic. Attempting to manage the allocation and deallocation of the trees in this piece of software would prove very challenging. When permuting an expression hundreds of instances of nodes are created and de-referenced. The garbage collector handles the memory de-allocation of these de-referenced nodes automatically.

JavaFX

When attempting to decide on the platform for creating the application Java Swing was initially considered. JavaFX is intended to replace the Java Swing library and seeing as its API is much cleaner and easier to work with it was chosen as the platform on which to build the Theorem Proving Assistant. For example, JavaFX takes care of alot of the mandatory code automatically such as redrawing the application after an update. JavaFX also supports CSS styling which is the other language used to develop with. (All 17 lines of it!).

GSON

In order to save and load theorems, initially a plain text approach was taken but the need for more information than just the expression was needed, such as a theorems derivation string. Google's GSON library was used to turn Java data-bean objects to JSON strings for saving theorem sets, and JSON strings to Java objects for loading theorem sets. A JSON string captures the properties of an object (in terms of the objects in object orientated design) in plain text.

JUnit4

JUnit is a unit testing framework for Java. Throughout this project it was used for regression testing to maintain the integrity of current features while adding functionality.

IntelliJ

IntelliJ is a market leading IDE. As a student the commercial version is available for free. It was chosen due to it's ability to generate needed code quickly and automatically as well as its very powerful refactoring tools. At one stage in the development process a lot of Lists to had to be migrated to Sets. IntelliJ made this very easy, automatically migrating effected methods, fields and variables.

Git and GitHub

Git's version control system facilitated the exploratory approach to building this software package. New features were developed on their own branch and only when the feature was fully tested and correct was it merged into the master branch. Several branches were abandoned during the development process, reverting back to the master branch to start an entirely different approach.

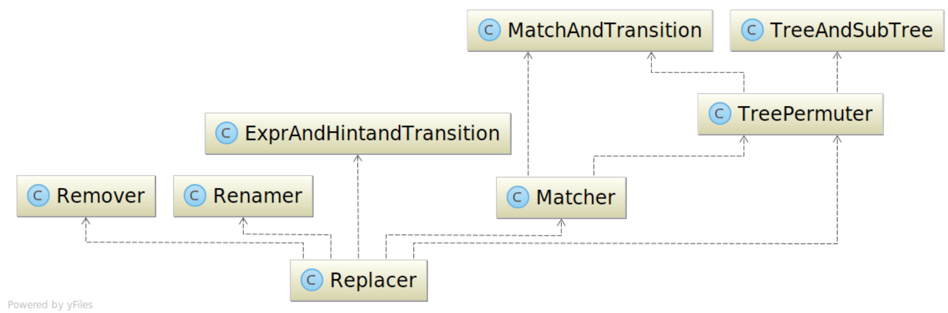
Github was used to back up the code base regularly and to share my code and application with others. Peers regularly provided 3rd party functionality testing and gave good feedback which influenced some design decisions.

Class Diagrams

The Theorem Proving Assistant consists of thirty seven Java classes across thirty six files. While it is possible to generate a diagram of the entire application it's so large it would offer no benefit. Instead three small subsets of the diagram are presented here.

The Backend Engine

The entry point to the engine is the Replacer class. It is the responsibility of this class, when given an expression, a selection and a rule, to yield possible replacement options. It directly relies on four other worker classes and indirectly on three data-bean classes.

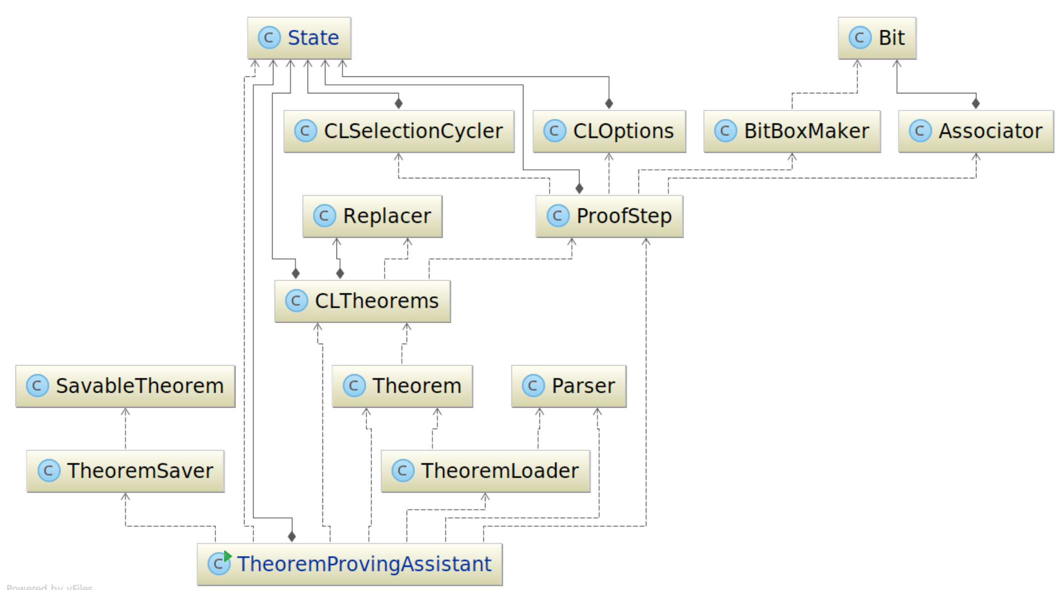


The Front End GUI

The TheoremProvingAssistant class is the entry point of the Gui. Any class pre-appended with CL is a click listener. The theorem click listener has a Replacer which is called upon to yield valid replacements.

The State class in the top left maintains the state of the application. References to values such as the current expression, the list of theorems available and the steps in the current proof are all available through this class. It is not a manager as it has no logic code, it merely holds references to values.

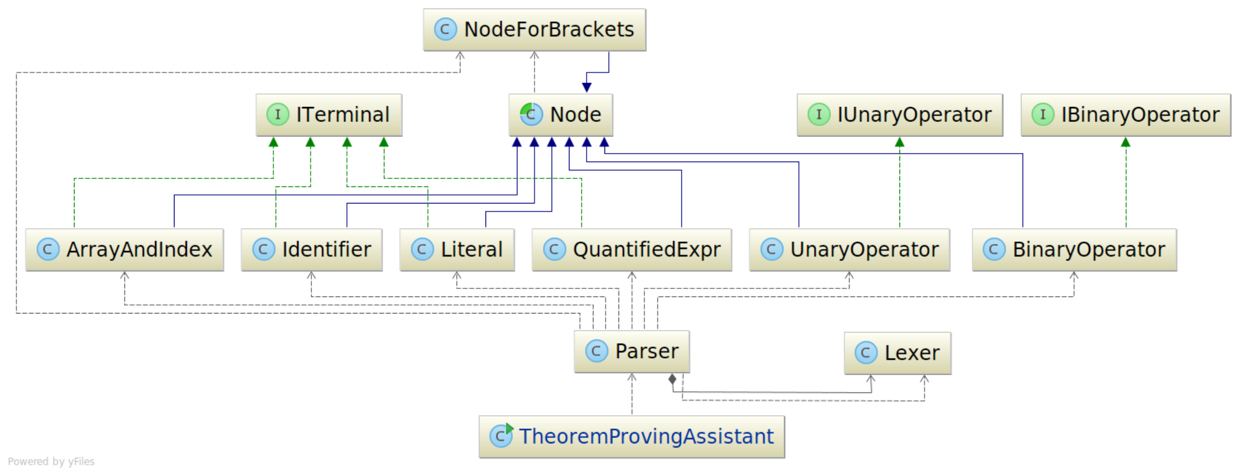
The Associator and BitBoxMaker classes can be seen depending on the Bit class in order to create a ProofStep. Both the TheoremProvingAssistant and TheoremLoader classes depend on the Parser to build syntax trees.



The Parser

It is the role of the Parser class to create syntax trees. These trees consists of Operators and Terminals. Code which is common to all nodes is implemented in the Node class, a superclass of the specific nodes.

Each non bracket-type node class implements either ITerminal, IUnaryOperator or IBinaryOperator. It is with these interfaces that the classes can be referenced polymorphically. For the same reason, although it is not displayed here for clarity, every class which is used as a node implements INode.



Testing And Evaluation

Regression Testing

Regression Testing is a type of software testing that ensures that functionality written and tested previously still works after it is amended, extended or interfaced with other features. These tests comprise of a suite of individual unit tests, each testing one discrete use case of a particular feature.

Throughout the build process it was important to maintain a suite of unit tests to verify the integrity of features as work continued. It is necessary to know that code being relied upon in a black-box manner is still functioning as expected. This regression testing increases the chances of bugs being detected throughout development.

For every new feature developed unit tests were written. These tests covered general use cases as well as extreme edge cases of the added feature. These tests were packaged into three major suites:

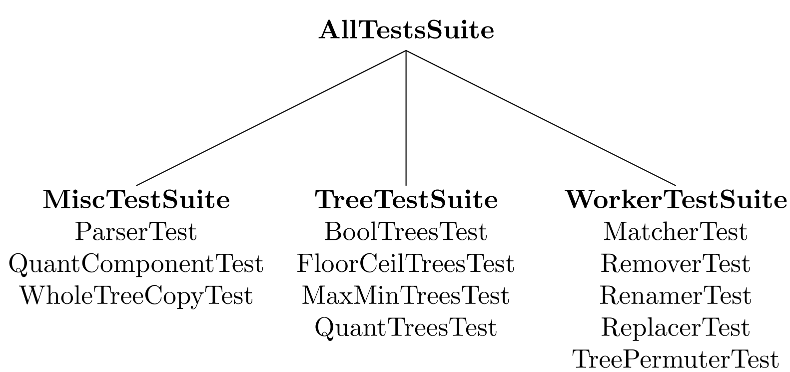
Tree Test
The trees are the core of this project. It is necessary to know that they are being walked and read correctly. This suite of tests walks and prints the characters in an in-order traversal (with the exception of bracket-type and unary operator nodes) and compares the output with an expected corresponding string.

Worker Tests
The worker tests were the most important tests. Throughout the build process the workers were treated as black-boxes. Their implementation wasn't known and they were relied upon to provide consistently valid results. Often the addition of some function to a worker resulted in tests failing in another. Maintaining a fully passing test suite enabled the desirable black-box style of programming.

Misc Tests
These test were for testing small independent features of the application such as the tree copying logic as well as testing the individual components of a quantified expression.

These three test suites can be executed independently or simultaneously with a suite that incorporates all three.

The test hierarchy is depicted below.

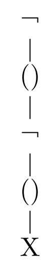


Complexity Analysis

The Theorem Proving Assistant uses depth first search (dfs) for tree traversals in all of it's algorithms. It's the primary method used of walking syntax tree's to find a node, compare whole trees or even just yield the expression represented by the tree. Due to the importance dfs it's time complexity and space complexity are discussed here.

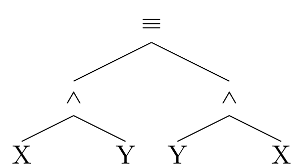
Time Complexity
Every node in the tree is visited once. Some operation on the node occurs at each visit so the time complexity of an algorithm using dfs for an in-order traversal will depend on the operation carried out at the visit. However, seeing as the number of visits is limited to the number of nodes in the tree, the fundamental depth first search traversal is $O(n)$ where n is the number of nodes in the tree.

Space Complexity
Almost all of the tree traversals in the Theorem Proving Assistant are recursive. For this reason each recursive call during a traversal will consume more memory on the call stack. In a fully unbalanced tree, where every node is exclusively a left (or right) child of it's parent, the depth of the tree will be equivalent to the number of nodes in the tree. A fully unbalanced tree is depicted below for the expression $\neg(\neg(X))$.



This worst case example will need n recursive calls to be pushed onto the call stack and thus, will have a space complexity of $O(n)$ where n is the number of nodes in the tree.

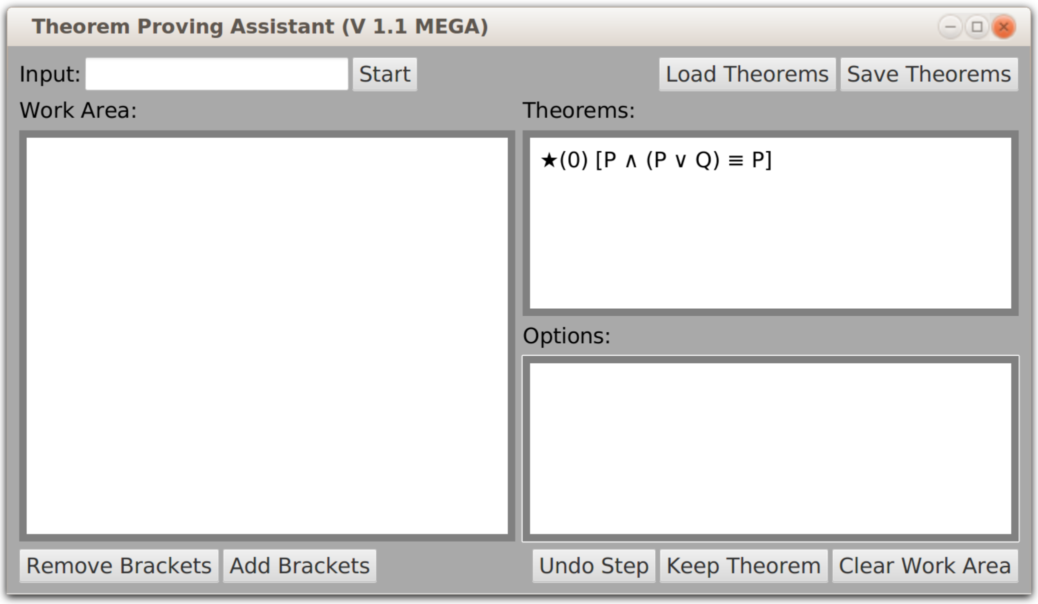
Below we examine the fully balanced tree for the expression $X \wedge Y \equiv Y \wedge X$. The maximum depth of any fully balanced tree is at most $\log(n)$ where n is the number of nodes in the tree.



In this best scenario the maximum number of calls on the call stack will be $\log(n)$ so the space complexity will be $O(\log(n))$.

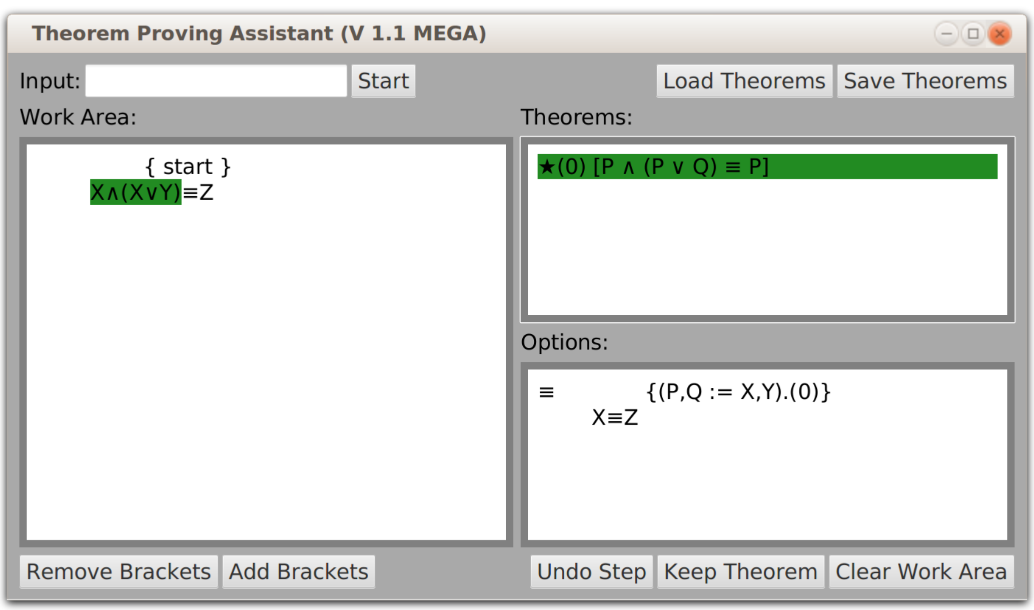
Proving Correctness: A Case Study

The regression testing is needed to verify that this algorithms are working as expected. Proving correctness of the software is an entirely different process. Here a case study over the boolean calculus is presented with simple, complex and invalid cases. We consider the application of absorption0 to a variety of expressions. Every test case begins with an empty work area and the single theorem, absorption0, loaded into the theorems area.
Some convoluted mappings are identified in the process which are technically correct but are not needed in this case study. They have been removed from the screen shots for clarity.



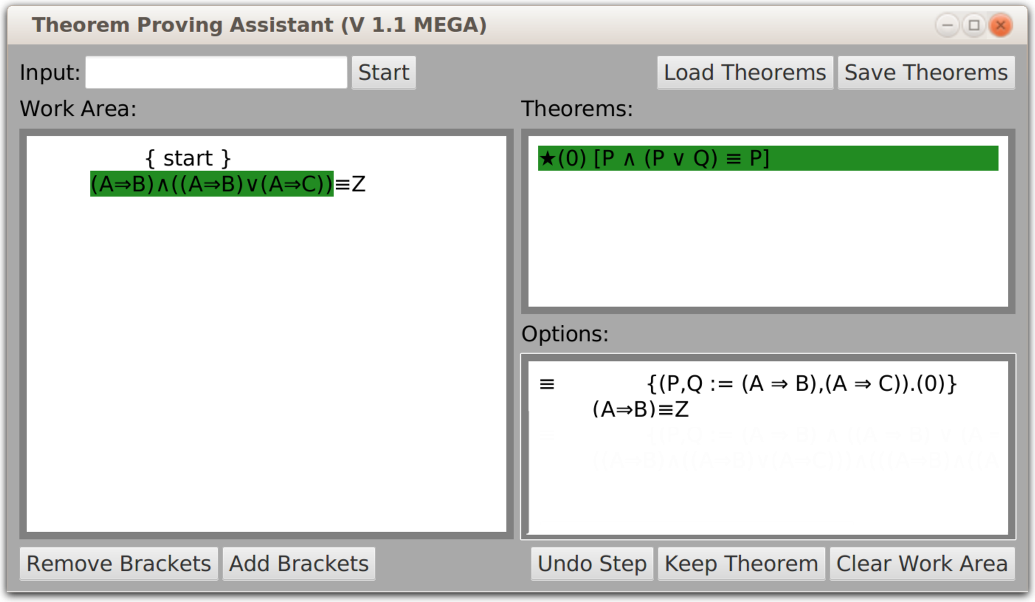
Case 1: A Simple Case

When absorption0 is applied to the expression $X \wedge (X \vee Y) \equiv Z$ the theorem proving assistant identifies a near exact match. The entire left hand side of the expression maps directly to the left hand side of absorption0. The expected mapping of $P, Q := X, Y$ is applied to the rule and the resulting expression $X \equiv Z$ generated in the options area.



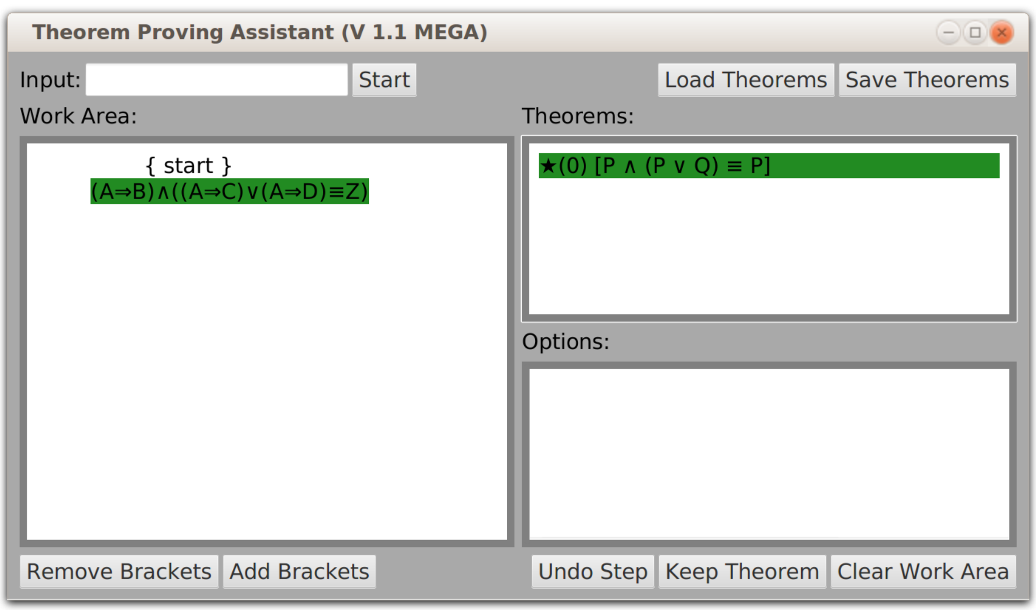
Case 2: A More Complex Case

In this case a much more complex expression is being used, $(A \Rightarrow B) \wedge ((A \Rightarrow B) \vee (A \Rightarrow C)) \equiv Z$. The mapping no longer needs to map an identifier to an identifier but an identifier to an entire expression. The Theorem Proving Assistant identifies the need for $P, Q := (A \Rightarrow B), (A \Rightarrow C)$ and correctly applies the mapping to the rule to yield $(A \Rightarrow B) \equiv Z$.



Case 3: A Case Which Should Generate No Options

This case correctly generates no options for the expression $(A \Rightarrow B) \wedge ((A \Rightarrow C) \vee (A \Rightarrow D)) \equiv Z$. This is a very subtle change to the expression used in case 2. While the operator pattern of the expression matches that of the rule, the Theorem Proving assistant has identified the need for the first and second term in the rule to be the same. That constraint does not hold for this expression as the first and second term are $(A \Rightarrow B)$ and $(A \Rightarrow C)$ respectively.



Conclusion

The Algorithms

Pattern matching algorithms to perform theorem proving in the style used by Djysktra were developed. The most important of these was the tree permutation algorithm. It manipulates a syntax tree to yield different versions of that tree while maintaining the expression yielded after an in-order traversal. This algorithm was relied upon when generating lists of replacement options for a given sub-expression and a given rule. Many small worker algorithms were employed during this process such as matching, renaming and replacing.

A further set of algorithms were devised to link the replacement algorithms with a user interface. When generating a string with which the user can interact with that string is associated with a set of syntax trees. The association algorithms allow a user to select sections of that string while at the same time (unknowingly) select a subtree of one of the trees in the set.

The Software

All of the algorithms devised were implemented in Java and linked with a graphical user interface built on the JavaFX platform. This interface, called the Theorem Proving Assistant, provides a rich environment for theorem proving over several calculi. The user can start a proof with an expression input in plain text, and then by selecting sub-expressions of that input and a theorem to apply can choose replacement options. In this manor the proof develops, one step at a time until a conclusion has been drawn and the user chooses to keep that theorem. Theorem set files can be loaded, saved and even written or amended in plain text.

The Theorem Proving Assistant has been proven complete, correct and robust by constant regression testing during development and conclusive case studies after development.

Further Work

As it stands the theorem proving assistant is restricted to the operators that have been hard coded into it. The next logical step in it's development would be to break this restriction. I like the idea of presenting this piece of software as a blank canvas with no pre-loaded operators. This would make the software package completely generic.

A means would need to be implemented to input a calculus's operators with relevant information such as the order of precedence, what an opening and closing bracket looks like, is it unary or binary and what the opposite of none commutative operators (such as implication and follows from) look like. Currently this information is hard coded into static data structures. As well as implementing a means of new operator input input these static data structures would have to be migrated to dynamic and handled accordingly.

Final Thoughts

I started building this project as a tool for use in teaching predicate calculus theorem proving. It was exciting to build something that (as far as I'm aware) did not exist yet. The further into the project I got, the more I realised that my thoughts and motivations were narrow minded. I started to read about pattern matching and it's relationship with intelligence. I found it fascinating to abstract from understanding mathematical expressions in terms of value and end results, but as a pattern and shape. It relates to so many aspects of life. When we, as humans, approach a three lane junction that we've never been to before we don't fail. We've been to junctions with similar patterns before. We have a rule for how to handle this situation. We rely on this knowledge to make educated decisions, not guesses, about how to act.

This project has led me to believe that pattern matching is a fundamental element of intelligence and has sparked my interest in it. I would love the opportunity to explore this further than the scope of the Theorem Proving Assistant. I would love to explore the potential of my algorithms in other fields, finding other applications for pattern matching software for calculational purposes.

Appendix

A: The Plain Text Pseudo-Language

Boolean Expressions

$X \equiv Y$: `"X == Y"`
$X \not\equiv Y$: `"X !== Y"`
$X \wedge Y$: `"X and Y"`
$X \vee Y$: `"X or Y"`
$X \Rightarrow Y$: `"X -> Y"`
$X \Leftarrow Y$: `"X <- Y"`
$\neg X$: `"!  X"`
$X \wedge (Y \vee Z)$: `"X and ( Y or Z )"`

Quantified Boolean Expressions

$\langle \forall i : r.i : f.i \rangle$: `"<| forall i :  r.i :  f.i |>"`
$\langle \exists j : s.j : g.j \rangle$: `"<| exists j :  s.j :  g.j |>"`

Floor/Ceiling Expressions

$\lfloor x \rfloor$: `"|_ x _|"`
$\lceil x \rceil$: `"|' x '|"`
$\lfloor x \rfloor \leqslant x$: `"|_ x _| <= x"`
$\lceil x \rceil \geqslant x$: `"|' x '| >= x"`
$\lfloor x + n \rfloor = \lfloor x \rfloor + n$: `"|_ x + n _| = |_ x _| + n"`

Lattice Theory Expressions

$\langle \forall x, y :: x \sqsubseteq y \wedge y \sqsubseteq x \Leftarrow x = y \rangle$: `"<| forall x,y :  :  x under y and y under x <- x = y |>"`
$\langle \forall z :: x \sqsubseteq z \equiv y \sqsubseteq z \rangle \Rightarrow x = y$: `"<| forall z :  :  x under z == y under z |> -> x = y"`

Max/Min Expressions

$x \uparrow y$: `"x up y"`
$x \downarrow y$: `"x down y"`

B: The Grammar

```
Expr := NotEq { <EQUIV> NotEq }
NotEq := Impl { <NOT_EQUIVAL> Impl }
Impl ::= FF { <IMPL> FF }
FF ::= Or { <FF> Or }
Or ::= And { <OR> And }
And ::= Equals { <AND> Equals }
Equals ::= NotEquals { <EQUAL> NotEquals }
NotEquals ::= Lt { <NOT_EQUALS> Lt }
Lt ::= Gt { <LT> Gt }
Gt ::= Lte { <GT> Lte }
Lte ::= Gte { <LTE> Gte }
Gte ::= Over { <GTE> Over }
Over ::= Under { <OVER> Under }
Under ::= Up { <UNDER> Up }
Up ::= Down { <MAX> Down }
Down ::= Add { <MIN> Add }
Add ::= Minus { <PLUS> Minus }
Minus ::= Factor { <MINUS> Factor }
Factor ::= <ID> | <NOT> Factor | <LPAR> Expr <RPAR> | <LFLOOR> Expr <RFLOOR>
           | <LCEILING> Expr <RCEILING> | <ARRAY_AND_INDEX>
           | <LANGLE> <EXISTS> <ID> <COLON> Expr <COLON> Expr <RANGLE>
           | <LANGLE> <FORALL> <ID> <COLON> Expr <COLON> Expr <RANGLE>
<EQUIV>  ::= "=="
<NOT_EQUIV>  ::= "!=="
<IMPL>  ::= "->"
<FF>  ::= "<-"
<OR>  ::= "or"
<AND>  ::= "and"
<EQUAL>  ::= "="
<NOT_EQUALS>  ::= "!="
<LT>  ::=  "<"
<GT>  ::= ">"
<LTE>  ::= "<="
<GTE>  ::= ">="
<OVER>  ::= "over"
<UNDER>  ::= "under"
<MAX>  ::= "up"
<MIN>  ::= "down"
<PLUS>  ::= "+"
<MINUS>  ::= "-"
<ID> ::= "[A-Z,a-z]+"
<NOT> ::= "!"
<LPAR> ::= "("
<RPAR> ::= ")"
<LFLOOR> ::= "|_"
<RFLOOR> ::= "_|"
<LCEILING> ::= "|'"
<RCEILING> ::= "'|"
<ARRAY_AND_INDEX> ::= "[A-Z,a-z]+\.[A-Z,a-z]+"
<LANGLE> ::= "<|"
<RANGE> ::= "|>"
<COLON> ::= ":"
```

References

[1] IBM Icons Of Progress - http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/

[2] Dominic Basulto 2016 - http://bigthink.com/endless-innovation/humans-are-the-worlds-best-pattern-recognition-machines-but-for-how-long

[3] 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.

[4] Edsger W. Dijkstra - https://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1240.PDF

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

[6] 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.