Proof Assistant Documentation

Declan Thompson

Version 4.0
28th February 2014

Contents

1 Introduction

This document aims to explain how the proof assistant works from a programming perspective. It has been written in an attempt to facilitate understanding of the code, in case anyone wants to change, add or fix anything.

I will begin by giving an overview of how the proof assistant works in general. This will be followed by an explanation of the major classes and their functions. I will give general directions on adding more rules. Finally, I will present a series appendices which may be of use to anyone hoping to change the program.

I have written proof assistant over a number of months and my algorithms for completing various tasks have been refined as I go. As such, you may be horrified by the apparent complexity of early-implemented rules (like equivalence elimination), especially compared with the (comparative) elegance of the later rules (like the box rules). You are also likely to come across a huge amount of superfluous code. Some of this forms remnants of previous algorithms, some is in preparation for an (as yet) unimplemented feature, some is and always has been useless.

1.1 To the Reader

I am writing this document in the hope that it is of use. Unfortunately, I am running out of time to provide complete and all-encompassing documentation. As such, there may be errors and grammatical/spelling mistakes within this work. Please excuse them. I created an initial version of this document a number of months ago. It was an incomplete but close description of the proof assistant at that time. It is now out of date but may still contain some useful information. For this purpose, I have included it as Appendices C onwards. Use this information with extreme caution.

2 Overview

2.1 Driving Ideas

What a horrible section title.

In this section I hope to give you an idea of how I intend the proof assistant to work overall. It currently does not quite live up to this ideal, but it would be nice if any changes took it into account.

2.1.1 Proof Methods

This class should be where the proof itself is stored and manipulated. All the rules should be present as methods, as well as their helpers. This class should be the "backend" of the proof assistant - the user interface can interact with it, but the user never has direct access to the Proof Methods class.

This allows for the possibility of a variety of user interfaces, all with the same underlying functionality. I have barely looked into it, but this may allow for easier implementation of an Android app, for example.

While this is a nice theory, I have not adhered entirely to it. In particular, I use Swing dialogs to request further information from the user, or to present errors.

2.1.2 Proof Panel, Proof Frame

The proof panel and proof frame are implementations of an interface. The proof panel was designed to match as closely to the pen and paper proof method as was reasonable. For this reason, lines a give specified places and a layout manager is not used.

The interface has been designed to require as few clicks from the user as possible. Prettiness has come far behind functionality and simplicity.

2.2 Human Readable and Longer

Here I'll try to explain how the proof assistant actually works.

2.2.1 NDLines and The Proof Itself

The NDLine class defines the NDLine object. These represent a line of a proof. There is a lot of information contained within each NDLine and a massive number of methods. The main pieces of information contained, however, are the line number, the line contents and the justification.

The line number is a non-zero integer. A positive line number will be displayed on screen. A negative line number will not. If a line number is negative, but the NDLine has a SpecialNum, then that will be displayed as the line's line number. This is to accommodate axioms.

The line contents is the actual formula, *stored in TEX code format*. From this formula, the main operator, first argument, second argument, parsed line and more are all divined.

The justification contains an NDJustification object. This will be one of the numerous Just....

NDLines are also given a type, which can be specified when they are created. Types are used to determine, among other things, which justification to create the line with, the contents of the line and the location of scope lines. A summary of the types can be found in Table 1.

The proof itself is stored as an array of NDLines in the Proof Methods class. The index before an unjustified NDLine should always be a blank NDLine. The proofArray is stored in the Proof Methods class, as well as in the Globals class. Be careful - the array in the Globals class is not always up to date.

2.2.2 Applications of Rules

Rules are all present as methods in the Proof Methods class (hence the name). Rules are given a goal and a resource. They act upon the proofArray as required by the rule, creating a replacement array that matches the rule's result.

Note that the rules themselves are rather dumb - they do not consider what operator the goal/resource has. Rather, the apply their rule, assuming the goal/resource has the correct operator. It is up to the interface to ensure that a goal with main operator & does not have negation introduction applied to it.

2.3 Technical and Brief

ProofAssistant creates a new ProofFrame.

If proofArray exists, ProofFrame creates a new ProofPanel. ProofFrame provides general options. If a user creates a new proof, a new ProofPanel is created and Globals.assist is instantiated to a new ProofMethods.

ProofMethods handles the proof itself. Any rule applications occur in Proof-Methods. It provides the TeX code output and the text output.

ProofPanel displays the proof and handles user interaction.

NDLine represents a line in the proof. It contains methods for finding the main operator of the line, the arguments, parsing the line, getting the line number and getting the line justification.

NDJustification represents a justification for a line. The base class provides accessor methods. NDJustification is extended for each type of justification, to set the required parameters.

3 Major Classes

In this section, I will explain the features of the major classes, and going into detail in some of the tricker areas.

3.1 ProofAssistant

This class contains the `main` method of the application. It sets up the GUI.

The `createAndShowGUI()` method does as it's named and deals with the differing types of command line arguments that may be supplied. The method creates a new `ProofFrame` and, in one case, a new `ProofMethods`.

3.2 ProofFrame

This class extends `JFrame` and creates the main application window. Only one of these is created, and it is assigned as `Globals.frame`.

The main content area of the frame is given a `BorderLayout`, with a `ProofPanel` in the centre and a `StatusBar` south. The `myInit()` method sets the look and feel to the system default, creates file choosers and sets the size and location of the window. The `reviewSetup()` method reviews the `.config` file, which is created by the `saveSetup()` method.

The `inputIsGood` and `lineInputIsGood` methods check that inputted phrases are correct before they are passed to other classes.

The setListOfSystems method adds the default presets to the preset list. It should probably be moved out of ProofFrame to somewhere more relevant (Globals perhaps?).

The remaining methods are (hopefully) fairly straightfoward. actionPerformed deals with events from the menu. If you change anything with "newProof", be sure to change it with "newProofFromTeX". I've customarily used "aboutApp" as a place to test out new features. actionPerformed is also where the file opening and closing occurs for Save and Open. saveProof and openProof basically mirror each other. They should be fairly straightforward, but quite long.

3.3 ProofMethods

Herein lies the guts of the application - this is where the deduction actually happens. There are two constructors - the one that takes a String[] is the most frequently used - the other is used only when opening a proof. As you will see, they are almost identical.

The setRulesAllowed method does nothing it would seem. readInArgs, however, creates the proofArray itself, including any extra lines. There are a large number of methods used as helpers to the numerous rules present here.

3.3.1 General Rule Structure

As you will note, all rules take two arguments - a goal and a resource. I have set the methods up like this for consistency. Even though some methods (like conjunction introduction) only use one of their arguments, both are required. If a current resource is not selected, the current goal is supplied as both arguments.

3.4 NDLine

3.5 Globals

The Globals class was born out of a desire to have a global line number variable, so that each newly created NDLine would have a sequentially increasing line number. A lot has been put into the class since, and it should probably be converted to use non-static methods, and have instances.

4 Adding More Rules

Herein I attempt to present a guide to adding new rules. The steps do not need to be done sequentially, but doing so will avoid the error messages many IDEs give if a method does not yet exist. I make limited attempts to explain the purpose of each step.

This guide assumes you are creating both introduction and elimination rules for a new operator. If this is not the case, you should be able to adapt the guide to your needs.

4.1 Initial Considerations

Before adding a rule, you will need to consider a number of things.

First, you will need a unique name identifier for your operator. Conjunction is assigned the identifier `con`, as an example. For a list of the current name identifiers, see Table 2. This identifier will be the name used for correctly parsing a symbol and selecting a rule. By convention, I have used this name in rule method names, for example `conIntro` and `conElim`.

Second, you will need a symbol to use for the operator. This is what will be displayed onscreen, and what you will be required to input in the New Proof dialog.

Finally, give some consideration to what "category" of rules this fits (Basic Rules, Special Rules, Hybrid Logic...).

4.2 Additions in Globals

You will need to make two additions in the Globals class.

Under `setDefaultOps()` and `setNonAucklandOps()`, add an appropriate line. These assign characters to the operator names. The format of the lines to add should be abundantly clear.

Under `setDefaultRulesAllowed()` and `setRulesAllows`, add appropriate lines. These methods govern changes in rule presets. The rules set in `setDefaultRulesAllowed()` are by convention those supported by NJ.

4.3 Additions in Proof Methods

Add appropriate methods for elimination and introduction in an appropriate section. You may wish to create a new section.

If you wish for "pointers" on how a rule can be structured, look to the existing introduction and elimination rules. The `else` of `atIntro` provides a nice template.

You also need to add lines under `mMIntroActions` and `mMElimActions`. These again should be straightforward. These are used to help Magic Mode to run.

4.4 Additions in Proof Panel

You need to add lines under `introActions` and `elimActions`, similarly to those in `mMIntroActions` and `mMElimActions`. You do not need to worry about the buttons themselves - they access the `operators` hashmap, and so the symbols here have been set in Globals.

4.5 Additions in the Rule Palette

I used Netbeans IDE to create the proof assistant, and used the GUI builder to create the rule palette.

You need to add a new checkbox (and possible a new category) for each new rule. Set its `itemStateChanged` as per the other `itemStateChangeds`. Set the checkbox's `setSelected` to `Globals.rulesAllowed.get(name)`, where `name` is the rule name, as specified in Section 4.2.

Add appropriate lines in `savePresetButtonActionPerformed` and `presetComboBoxActionPerformed`

4.6 Additions for Friendly Input

Friendly Input is the name I have used for the graphical input options (as opposed to the TEX input options). You will want to set your symbol up so that it can be parsed correctly.

There are two friendly inputs, FriendlyInput and FriendlyLineInput, however both use the same parser, which is located in MyOptionPane. In this class, under processSequent and processLine are two identical sections, setting up two HashMaps binOps and preOps. Depending on whether your symbol is prefix or infix notation, add it to the correct hashmap. You may wish to add special code in parseLine.

Both FriendlyInput and FriendlyLineInput also use the same SymbolSelectorPanel. If your symbol is not easily accessible on the keyboard, you may wish to add it here. You may also wish to add a keyboard shortcut. This can be achieved by adding appropriate lines under myInit in FriendlyInput and FriendlyLineInput.

A Useful Tables

In writing the proof assistant, I ended up with a number of pieces of paper scattered around upon which I relied. I preserve them digitally here.

0	Standard line
1	Assumption start line
2	Assumption end line
3	One line assumption
4	Premise
5	Blank Line
6	Used solely for find the first and second arguments
7	Proof Box start line
8	Proof Box intermediate line
9	Proof Box end line
10	Proof Box start line (equivalence introduction)
11	Axiom

Table 1: NDLine Types

B Proposed Further Features

There are a number of features that were not implemented due to lack of time and importance. However, I present them here in case some one has time in the future. I also give some thoughts on how they might be implemented.

B.1 Related to Proofs

Collapse identical lines Allow one of two identical lines in scope of one another to be removed. This will require updating the justifications of any lines justified by the removed line, which may take some doing. All the NDJustification extensions have mutator methods for line numbers, but

con	Conjunction
dis	Disjunction
imp	Implication
equ	Equivalence
neg	Negation
qe	Existential Quantifier
qa	Universal Quantifier
eq	Identity
box	Box (Modal Logic)
dia	Diamond (Modal Logic)
at	@ (Hybrid Logic)
nom	Nominal (Hybrid Logic)
self	@ Self-reference (Hybrid Logic)

Table 2: Name Identifiers

this would still be tricky. Furthermore, all line numbers greater than the removed line's number would need to be decremented.

Q7 in Q, not PA There are actually 7 Q axioms. The one missing from the proof assistant can be obtained through induction under PA. This feature would show the 7th if we are in Q but not in PA. This could be implemented as an extra line added after Q, similarly to how the Q axioms themselves are implemented.

Convert current goal to general case In some cases it is easier to prove the general case. This feature would allow the current goal to be converted to the general case (which would allow for induction, for example). The current goal would be justified by the general case line. It is possible to do this currently using Cut, but not automatically. Perhaps the term selector panel could be rigged such that the user selects which terms should be generalised.

Set the list for generated parameters Currently, parameters are generated from a list (described in the TermStore class). This feature would allow the user to set this list, and could be implemented in a similar method to the arity list options.

List of historical proofs This feature would maintain a list of the proofs a user has completed. At any point in a proof, they could bring up the list. It would should sequents relevant to the current proof. The user could then use these sequents as rules. The methods required for this would be like for implication elimination.

Line Shortcuts Some common types of line (like R is reflexive, R is transitive...) could be inputted in plain text. This has already (secretively) been implemented for some phrases in New Proof from TeX code. However, the lines do not show up as they might. Needs further development.

B.2 Related to Interface

Integrated Help I have already written a basic help file `proofassistanthelp`. This feature would add integrated help, I would suggest in the form of an onscreen character ("Fitchy" maybe) who would offer advice like "Looks like you're trying to prove the Law of Excluded Middle. Need some help with that?" and who would be animated, often tapping the screen or deforming into a bike. Or it could just be more tooltips.

Beamer Support Possibly entirely useless, this feature would be similar to the animated GIF output, but generate TEX code suitable for beamer.

More bracket options Currently, the user can choose to show no brackets or too many brackets. This option would allow the user to show an intermediate number of brackets.

Pinch to zoom This feature would add pinch to zoom support on touchscreens and trackpads. mt4j may be a good starting point.

Platform-specific distributables Create .app and .exe files. This will allow icons (!) and to associate .ndp and .ndu files with the proof assistant.

Method to browse and launch sequents The user would be able to browse through sequents in some format and then launch the proof assistant with the sequents loaded. This could (maybe) be done with links in pdfs - if .ndp and .ndu files are associated with the proof assistant, they could form the links. This method would tie in with the previous item.

B.3 Related to Code

Smarter parsing Allow a broader range of inputs. This task is never ending.

Tidy code I admit that it's pretty bad.

Instantiate Globals You'll notice that Globals is a static class. It might make sense to change this, so that more than one proof can be in play at once.

C ProofAssistant

This is the main class of the program. It contains two methods.

C.1 `main`

This method is called when the program starts. It creates an awt event queue which runs the other method in ProofAssistant, `createAndShowGUI`.

C.2 `createAndShowGUI`

This method is called by `main`.

It creates a new ProofFrame called `frame`, using the NDLine[] array Globals.proofArray. `frame` is set to be visible and Globals.frame is set to frame.

D ProofFrame

This class extends the JFrame class. It forms the main Proof Assistant window and has methods for dealing with menubar items and setting the Look and Feel of the program.

D.1 `ProofFrame`

This is the constructor method of ProofFrame. It sets the title of the frame to "Proof Assistant".

It then checks if the `proofArray` has been instantiated. If it has, it creates a new ProofPanel called `panel`. Note that initially, this class is called by ProofAssistant, and in that case `proofArray` is `Globals.proofArray`.

The size of the frame is determined based on `Globals.proofWidth` and `Globals.proofHeight`. The following lines ensure the frame appears in the middle of the screen when initialised.

Finally, the default close operation is set to exit, and `setMenus` is run.

D.2 `setLAF`

This method sets the look and feel of the program to the System look and feel. It is not called in version 1.0.

D.3 `setMenus`

This method creates the menubar `menuBar` for the frame.

D.3.1 `file`

A JMenu called `file` is created. This has the mnemonic F.

Three menu items are add to `file`. They are `newItem`, `exportItem` and `closeItem`. These are used as would be expected (`exportItem` exports the proof to TeX code).

D.3.2 `options`

A JMenu called `options` is created. This has the mnemonic O.

One checkbox item is added to `options`. This is `greyScopesItem` and is initially unchecked. This controls whether or not out-of-scope lines are greyed out.

D.4 `actionPerformed` and `itemStateChanged`

These methoda are called when a menu item is chosen. They carry out the required actions.

E NDLine

This class represents a line in a natural deduction proof.

An NDLine object is one of five types:

0 Standard Line
1 Assumption Opening Line
2 Assumption Closing Line
3 One line assumption
4 Premise
5 Blank Line

E.1 `NDLine`

There are three constructor methods.

The first takes a string `macro` and an integer `type` as arguments. This constructor can be used with any type of line. The `line` is set to `macro`, the `mainOp`, `firstArg`, `secondArg` and `parsedLine` are found. The `lineNum` is selected by incrementing `Globals.lineNum`. The `type` is set. The `type` is checked and an appropriate NDJustification created.

The second takes only a string `macro` as an argument. This constructor should only be used to create NDLines of type 0 (Standard Lines). The `line` is set to `macro`, the `mainOp`, `firstArg`, `secondArg` and `parsedLine` are found. The `lineNum` is selected by incrementing `Globals.lineNum`. The `type` is set to 0. A blank NDJustification is created.

The first takes only an integer `type` as an argument. This constructor should only be used to create blank NDLines (type 5). The `line`, `mainOp`, `firstArg`, `secondArg` and `parsedLine` are all set to "". The `lineNum` is set to 0. The `type` is set to 5. A blank NDJustification is created.

E.2 Private Methods

E.2.1 `findMainOp`

This method returns the main operator of the line. It takes a string `line` as its only argument.

The index of the first instance of "{" in `line` is searched for. If the result is -1 (i.e. the bracket is not found), "" is returned. Otherwise, a substring stripping the first character (which should be \) and stopping before the first { is returned.

E.2.2 `findFirstArg`

This method returns the first argument of the line, if it exists. If the first argument does not exist, it returns the empty string. It takes a string `line` as its only argument.t.

The method first checks whether or not the mainOp exists. If it does not, the empty string is returned.

Otherwise a bracket count is started. When the number of open brackets matches the number of close brackets, the end of the first argument has been found. The substring from the first open bracket to the relevant close bracket is returned.

E.2.3 `findSecondArg`

This method returns the second argument of the line, if it exists. If the second argument does not exist, it returns the empty string. It takes a string `line` as its only argument.

The method first checks whether or not the mainOp exists. If it does not, the empty string is returned. If the mainOp is "neg" (a unary operator), the empty string is returned.

Otherwise a similar bracket count to `findFirstArg` occurs. However the substring returned is from the relevant open bracket to the last close bracket of the line.

E.2.4 `parseTheLine`

This method returns the line, parsed to unicode output. It calls itself recursively and takes a string `line` as its only argument.

The main operator of `line` is found. If the line is exactly "\falsum", the line is sent to `convertOp`. If the main operator does not exist, the line is returned.

Otherwise, the main operator does exist. The method checks whether a second argument exists in the line. If it does not (i.e. the line has a unary operator), the method sends the operator to `convertOp`, applied `parseTheLine` to the first argument and returns the result.

If a second argument does exist, the method checks whether the main operator is a quantifier. If it is, the method sends the operator to `convertOp`, applies `parseTheLine` to the second argument, and returns a string conforming to the correct style.

If a second argument exists and the main operator is not a quantifier, the method sends the operator to `convertOp` and parses both arguments, then returns a string conforming to the correct style.

E.2.5 `convertOp`

This method returns the unicode character for a logical symbol. It takes a string `op` as its only argument.

The string is checked in each conditional. If it matches one, that value is returned. Otherwise, `op` is returned.

E.3 Mutator Methods

E.3.1 `setJustification`

This method takes an NDJustification `just` as its only argument. It sets the justification of the line to `just`.

E.4 Accessor Methods

E.4.1 `get...`

These methods behave as would be expected. They take no arguments and return and object conforming to the request.

E.4.2 `parseMainOp`

This simple method returns the main operator of the line, parsed to unicode as a string.

If the line is falsum, the line is sent to `convertOp`. Otherwise, the main operator is sent to `convertOp`. The result is returned.

E.4.3 `parseFirstArg()` and `parseSecondArg()`

These return strings containing their respective arguments, parsed.

E.4.4 `getTeX`

This converts the line to TeX code, using the ndproof.sty style. It checks the line type to ensure the correct LaTeX command is used. The final return statement should never occur.

E.4.5 `isInScopeOf`

This method returns true if the current line can "see" another line in a particular array of NDLines. It takes two arguments, the other line `anotherLine` and the array of NDLines `anArray`.

The index of the current line in `anArray` is found. A counter `scopes` is created, to be incremented if any assumption end lines are encountered. A boolean `inScope` is created and set to false.

Starting at the line above the current line, each line in `anArray` is inspected. If it is of type 2 or 3 (the close of an assumption), `scopes` is incremented.

If `scopes` is exactly 0 and the line matches the `lineNum` of `anotherLine`, `inScope` is set to true. That is, we have found `anotherLine` in `anArray` and it is in scope.

Finally, if the line is of type 1 or 3 (the open of an assumption), `scopes` is decremented, but not below 0.

The value of `inScope` is returned.

E.4.6 `indexIn`

This method returns the index of the current line in the NDLine[] array `anArray`. It takes an NDLine[] array `anArray` as its only argument.

The `lineNum` of the current line is found. This is compared against the `lineNum` of each index in `anArray`. If it is found to match, the index is returned. If no match is found, -1 is returned.

F Globals

This class holds variables to be used for global purposes. They are accessed by other classes using `Global.[variable name]`.

F.1 `lineNum`

This represents the line number of each line in the proof. Each NDLine object should have a unique `lineNum`, as this is what is used for finding the index of a line in an array of NDLines.

G ProofPanel

This class is the panel through which the user interacts with the proof itself. It extends JPanel.

G.1 `ProofPanel`

The constructor method takes one argument, an NDLine[] array `proofArray`. The layout manager of the ProofPanel is set to null.

`printLines` is called and a mouse listener is added to the panel.

G.2 `paintComponent`

This method is the paint component section of the class. It draws all the ScopeLines in the array `theScopes`.

G.3 `printLines`

This method prints the lines and buttons on the ProofPanel. This occurs in two sections.

G.3.1 Preparation for Printing Lines

In order that assumption scope lines may be printed correctly, it must be established how many scopes line will run concurrently. This information is also needed to determine the x-location for the line numbers and, consequently, the line contents and justifications.

The length of the longest line is required to determine the x-location of the justifications.

`proofArray` is looped through. At each index, the length of the line is checked against `longestLine`. `longestLine` is set to the maximum of these. The number of concurrent scope lines is counted, and `deepestJust` updated accordingly.

An array is created to hold the ScopeLines, the x-positions of the various labels is calculated and a stack is created to hold the indices of the scope beginnings.

H ExportFrame

I NDJustification

This class represents the justification of an NDLine. The class contains the constructor method and two accessor methods.

The constructor method takes no arguments and sets both `teXJustification` and `javaJustification` to the empty string. The boolean `blank` is left true.

The two accessor methods retrieve `teXJustification` and `javaJustification` as expected.

I.1 Just...

These classes extend the NDJustification class. They take various numbers of arguments depending on how what they are justifying. These arguments will always be line numbers.

Each class sets `teXJustification` and `javaJustification` to the relevant description, and sets `blank` to false.

Each class also includes mutator methods for setting the various line numbers.

J ScopeLine

K Documentation of Individual Features

K.1 New

The new feature allows a new proof to be created.

K.1.1 Overview of Implementation

The new feature is called from the menu item in the ProofFrame class. As such, most of the work is done within the ProofFrame class.

The user inputs a string to find the initial state of the proof. From this, a new ProofMethods is created, in `Globals.assist`. A new ProofPanel is created and the size of the ProofFrame set.

ProofMethods parses the string to an NDLine[] and prepares for the rest of the proof.

K.1.2 Implementation in ProofFrame

The JMenuItem `newItem` is declared and set in the `setMenus` method, where it is added to the `file` menu.

When the `newItem` is called in `actionPerformed`, a dialog box is shown to the user. This asks for sequents to be inputted. The input is sent to a string `s`.

If the string is of the sequent form, a type 6 NDLine called `temp` is created. This line is created solely to gain access to the `parseTheLine` method. An array called `tempArray` is created to hold the premises of the input. An array called `argumentArray` is created to hold the proof itself.

`tempArray` is iterated through. Each element has any initial spaces removed and is added to `argumentArray`. The remaining two spaces in `argumentArray` are filled by `-c` and the conclusion sequent.

The following occurs whatever the form of the input. The `Globals.lineNum` is set to 0, a new ProofMethods is created, and a new ProofPanel is created and set to the ProofFrame.

K.1.3 Implementation in ProofMethods

The ProofMethods class is called and takes a String[] as input. This input is immediately sent to the method `readInArgs`, the result of which the private variable `proofArray` is set to.

The `readInArgs` method iterates through the String[] of arguments. For all elements but the last two, a type 4 NDLine is created (a premise line). For the second to last element, a type 5 NDLine is created (a blank line). For the last element, a type 0 NDLine is created (a standard line).

K.1.4 Implementation in ProofPanel

The ProofPanel class is created with the `Globals.proofArray`, which has been set to the `proofArray` of ProofMethods.

The private variable of ProofPanel `proofArray` is set to the input `proofArray`. The layout of the ProofPanel is set to null, as we wish to place the items in specific locations. The `printLines` method is called and a mouse listener added to the panel.

The `printLines` method has reasonable complexity.

Initially, everything in the ProofPanel is removed. Four counters are initialised. These are used to determine the x positions of the various columns required for the proof. The `numScopes` counter is used in the creation of the ScopeLine[], which should not be needed when no assumptions are present.

The x positions for the line number column, the line contents column and the line justification column are calculated.

The `proofArray` is iterated through. If a line is type 5, it is ignored. Otherwise, for each line, labels `lineNum`, `lineContents` and `justification` are created, positioned, coloured and added.

If the index of the line is the same as the current goal's index, the text colour is set to blue and various rule buttons are added. If the index matches the current resource, the text colour is set to red. If `outOfScopeIsGrey`, the line is coloured grey if is it out of scope of the goal. Other lines are coloured black.

If a line is type 1 (ass start), a scope start index is pushed. If it is type 2 (ass end), the scope line is created.

Finally, `Globals.proofHeight` and `Globals.proofWidth` are set and the `paint component` is called.

The `paintComponent` method simply draws each element of the ScopeLine[] `theScopes`.

K.1.5 Implementation in NDLine

The NDLine constructor is overloaded to deal with different types of NDLines.

The most specified constructor takes a string `macro` and int `type` argument. The `macro` must be in the form outlined in `mylogicv02`. This is set to `line`. `line` is parsed for the main operator and the first and second arguments. The line is also parsed for java output (unless it is type 6 - which is used solely for finding first and second arguments). A justification is created depending on the type of the line.

The next constructor takes only a string as its argument. This constructor creates a line of type 0.

The third constructor takes only an integer as its argument. This constructor creates a blank line of the type supplied. It should only be used when 5 is supplied as the argument.

K.2 Undo

The undo feature allows for steps in a proof to be undone. It can be used to reverse any rule use in the proof.

K.2.1 Overview of Implementation

The undo command is called from the menu item, and so the bulk of the feature is implemented in the ProofFrame class. Here, the menu item is added and the action performed commands are presented. The basic implementation is as follows.

Globals contains four stacks for use in the undo feature, `proofsForUndo`, `goalsForUndo`, `resourcesForUndo` and `lineNumsForUndo`. When a rule is applied in the ProofPanel class, the current state of the proof is pushed to these stacks.

When the undo feature is activated (from within ProofFrame), the top item of each stack is popped and used to recreate the proof as it was one step earlier.

This implementation allows for infinite undos (while previous steps still exist).

K.2.2 Implementation in ProofFrame

The JMenuItem `undoItem` is declared at the beginning of the ProofFrame class as a private variable. It is initialised as a JMenuItem in the `setMenus` method, where it is given the shortcut key Ctrl+Z. The stack `proofsForUndo` is checked to see if it is empty. This result is used to determine whether or not `undoItem` is enabled.

When `undoItem` is selected, the top NDLine[] on the `proofsForUndo` stack is popped, and named `prior`. `Globals.assist` is set to a new ProofMethods using `prior` as a starting point. The `Globals.currentGoalIndex`, `Globals.currentResourceIndex`, `Globals.lineNum`, `Globals.proofArray` and the `panel`'s `proofArray` are set similarly.

Finally, the `panel` is refreshed using `panel.printLines()` and the enability of the `undoItem` is set.

A public method is also included in the `setUndoable` method. This method sets the state of the enability of `undoItem`.

K.2.3 Implementation in ProofMethods

To aid in this implementation of undo, the ProofMethods constructor was overloaded to allow construction using either a String[] (as always) or an NDLine[].

K.2.4 Implementation in ProofPanel

The implementation of undo in the ProofPanel class occurswithin the `actionPerformed`, `introActions` and `elimActions` methods. Exactly the same process occurs within each.

In `actionPerformed`, the undo implementation occurs in the `sameLineRule` conditional, under the heading `Prepare for Undo`. In the `introActions` and `elimActions` methods, the undo implementation occurs first, under the same heading.

The undo implementation first creates a new NDLine[] from the `proofArray` by cloning each item in that array. It pushes the result to `Globals.proofsForUndo`. The current goal index is pushed to `Globals.goalsForUndo`. If the current resource index is identical to the current goal index, -1 is pushed to `Globals.resourcesForUndo`. Otherwise, the current resource index is pushed to `Globals.resourcesForUndo`. This is due to a conditional in the `introRule` section which makes the current resource index identical to the current goal index if no resource has been selected. Finally, the current line number is pushed to `Globals.lineNumsForUndo`.

K.2.5 Implementation in NDLine

A `clone` method has been implemented in NDLine to aid in the implementation of undo. Because Java copies references when assigning variables, the targets are able to change even when the specific variables are not called. In an early test of the implementation, undo would not step justifications back, because the copied NDLines had changed their justifications. Hence, a clone feature has been implemented.

To summarise the above paragraph, Java does not include deep copying, however that is required for the implementation of undo. Hence, a `clone` method has been implemented to facilitate this.

The `clone` method creates a new NDLine called `theClone`, with the same `line` and `type`. The justification is set to the same as the current NDLine, as is the line number. Finally, `Globals.lineNum` is decremented, as it has been incremented during the creation of `theClone`.

The method `setLineNum` was also create to facilitate the undo implementation.

K.2.6 Implementation in NDJustification

For the same reasons outlined in the previous section, a clone method has been implemented in NDJustification.

This method creates a new NDJustification with the same properties as the current NDJustification.