uProve

<http://uprove.udd.be>

1. Introduction

uProve is program for building natural deduction proofs in propositional logic. The user builds the proof by selecting the lines to which s/he wants to apply a rule, and then select the rule in mind from a list of suggestions presented by the program. These steps are repeated until the proof is completed.

The program has been developed as a term project for the course Principles of Formal Software Development at the University of Ottawa.

This document's objective is to describe how the program is designed and implemented, how it can be installed and how it should be used.

2. Natural deduction

The style of natural deduction used in uProve is the one presented by Graeme Forbes in *Modern Logic* [1], but first developed by Gerhard Gentzen. The main idea is that a proof consists of a list of proof lines. Each one of those lines consists of a list of dependencies, a line number, a formula and a justification. The last line in a completed proof must contain the conclusion as its formula and only lines marked as premises in its list of dependencies. A complete list of inference rules in this system is provided in the end of this report.

A simple proof might look like this. The list of dependencies is written to the far left, followed by the line number within parentheses, then the formula and lastly the justification.

1	(1)	p->q	Premise
2	(2)	~q	Premise
3	(3)	p	Assumption
1,3	(4)	q	1,3 ->E
1,2,3	(5)	_|_	2,4 ~E
1,2	(6)	~p	3,5 ~I

The symbols used in uProve differ slightly from the ones used by Forbes. A list of symbols is shown below.

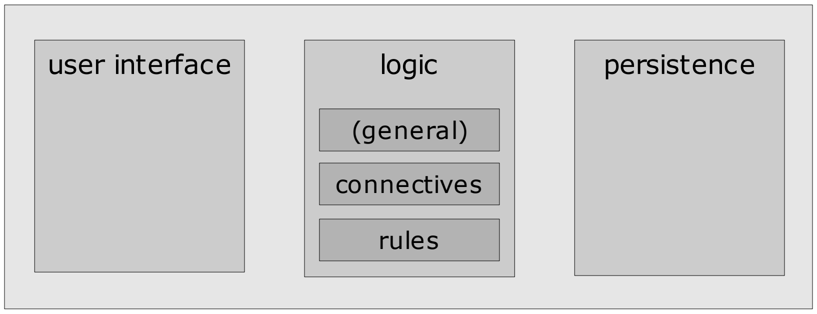
[?]	Placeholder, used to represent any formula
~	Negation
->	Implication
<->	Equivalence
\/	Or
&	And
_|_	Contradiction

3. Development

uProve is implemented in Java SE 5.0 (also known as 1.5.0)[1]. No external libraries except the standard ones are used. The build tool Apache Ant[2] has been used for simplifying the build process, and JUnit[3] has been used to run unit tests. However, neither of the two latter tools is needed in order to merely run the program.

3.1 Architecture and design

The program is built on a three-layer architecture, having a user interface layer, a logical layer and a persistence layer. The first and the third layer are, in Java, represented as one package each. The logical layer however is contained in three Java packages. One package consist of connectives, one of rules and one of general business logic.



These packages are not as stand-alone as I would have wanted. Some, but limited, business logic is placed in the user interface package to avoid complex workarounds. I however do believe that the design could be improved without making it more complex, but the problem is that I am not literate enough in the topic.

3.1.1 Internal representation of a proof

My object oriented design organizes proofs in a similar way as they are in the real (mathematical) world. That is, a proof (class *Proof*) consists of a number of proof lines (class *ProofLines*), each of which has a list of dependencies, a line number, a formula and a justification.

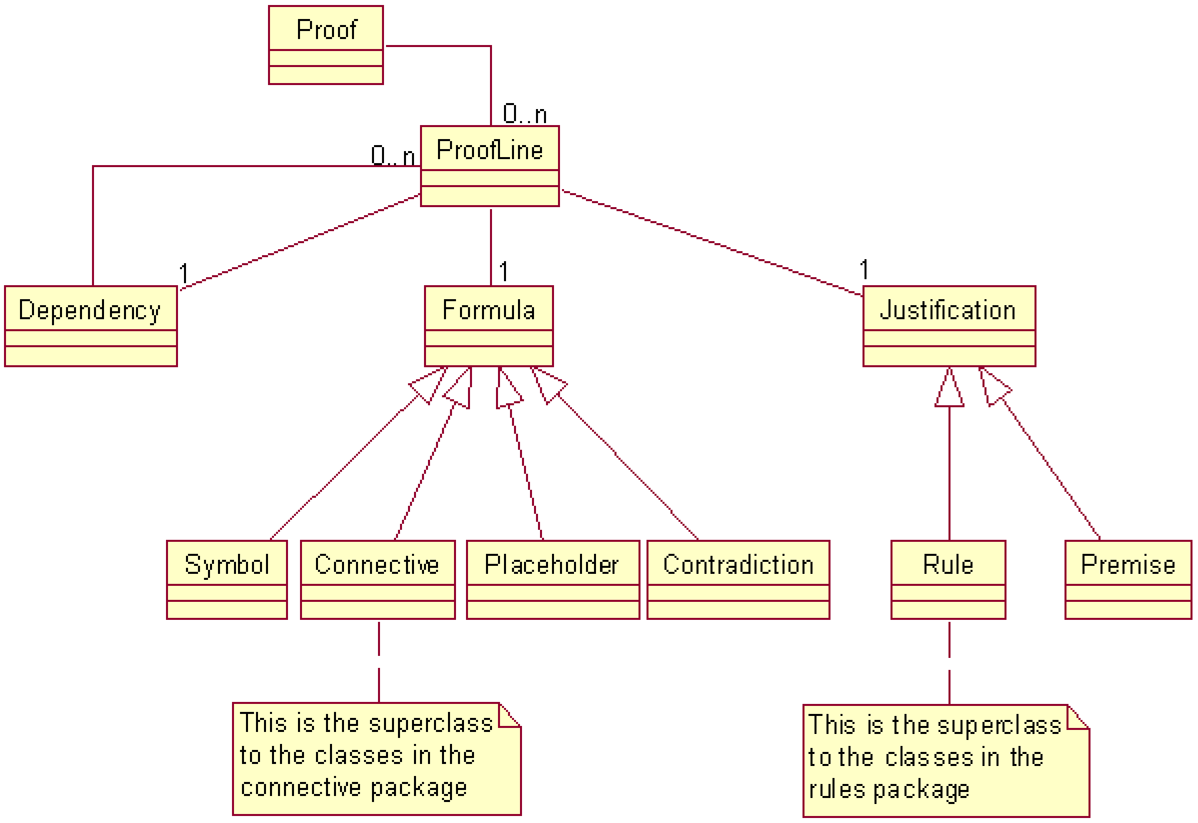
The list of dependencies, represented by the *Dependency* class, contains a list of proof lines.

The line number is represented as the primitive data type *int*.

The formula (class *Formula*) can take different forms, as seen on the drawing below. It can either be a symbol (e.g., p), a connective, a placeholder (i.e., [?]) or a contradiction (i.e., _|_). However, a connective alone does not form a well-formed formula. This is solved by letting the actual connectives contain one or two formulas themselves.

Finally, the justification can either be a premise (class *Premise*) or one of the Rules (class *Rule*).

1 Java can be downloaded from <http://java.sun.com/>.
2 Apache Ant can be downloaded from <http://ant.apache.org/>.
3 JUnit can be downloaded from <http://www.junit.org/>.


Proof
0..n
ProofLine
0..n
1
Dependency
1
Formula
1
Justification
Symbol
Connective
Placeholder
Contradiction
Rule
Premise
This is the superclass to the classes in the connective package
This is the superclass to the classes in the rules package

3.1.2 Construction of the list of suggestions

The most interesting part in the program is probably how the list of suggestions is constructed. When a user clicks a line, that line is packaged together with other selected lines into a *ProofLine* array. A *RuleTester* object is created to which the array is sent. The *RuleTester*'s responsibility is to instantiate all available rules, asking each of them to return all suggestions it has. The results, also in the form of *ProofLine*s, are then showed to the user.

The method handling the *RuleTester*'s request is implemented in the *Rule* superclass and not overridden in any of the rules. The method works in such a way that it first checks if the number of *ProofLines* is correct. If it is, it uses the *Permutator* to get all possible permutations of the input lines. Each combination is then checked to see if the rule can be applied. If it can, the resulting lines are collected and later sent back to the calling *RuleTester*.

At first I thought that this method would be too slow, since it does not use any intelligence in how the lines are reordered. It simply tries all possible combinations without considering what the lines look like. This is clearly not a problem for rules like implication elimination which only takes two input lines. However, I expected or elimination to be trickier. Five lines are required to do an or elimination, and those five lines can be ordered in $5!=5\cdot4\cdot3\cdot2\cdot1=120$ different ways.

I decided to try to this approach first, since the idea of having all rules being tried in the same way is appealing, but still design the system so it would be easy to override this method for individual rules if it would be necessary.

Once implemented, it turned out that these 120 checks was made so fast that it did not cause any problem whatsoever. The central implementation was therefore kept.

3.1.3 Copy rule

When implementing the functionality mentioned above for building the list of suggestions, I realized that I had to include one more rule than the ones presented by Forbes in order to not to violate the completeness of the system. In the system presented in the book it is acceptable to refer to the same line multiple times when applying a rule. However, this is not possible in uProve since a line only can be either not selected or selected, it can not be selected twice. This problem was solved by adding a simple copy rule to the set of rules available, making it possible to first copy a rule and then select both the original and the copy.

3.1.4 Plug-in system

I originally planned to make a plug-in system where rules (and perhaps even connectives) could be added or removed as wanted. Among other things, this would facilitate a way for switching from classical to intuitionistic logic by only removing one rule. The major benefit would however be that derived rules could be added by users themselves by just adding the corresponding class file.

I rejected this idea at a later stage when I realized that this plug-in system had some drawbacks. The first, and without doubt the most sever, is that this would impose security issues. Nothing would in this system stop a malicious person from writing a “rule” that would delete all files on the computer on which it is running. Since all derived rules used in a proof would have to be serialized and saved along with the proof itself, this would give a possibility to embed a virus in a proof. (This issue can be resolved by using the built-in security handling in Java, but other solutions to provide the same functionality are probably more suitable.)

3.1.5 Unreliable user interface

One major problem exists in this program, and that is the user interface. The functionality of the program is not affected by this, but it substantially reduces usability. The most common problem is that components changes size in ways they are not supposed to. It is recommended to have the program maximized in order to avoid these problems as much as possible.

3.2 Unit tests

My aim was to use test-driven development when developing this program. I did so for the logical layer but not for the user interface layer or the persistence layer. The components introduced in the logical layer due to these two other layers, for example the class responsible for handling the user's input, is not unit tested either.

In total, 227 unit tests have been developed and I believe that these have helped in developing the program and that they will be very valuable when modifying this program in the future.

3.3 Size statistics

The following table lists the different packages and how many lines of code each of them consists. (One line of code is here defined to mean one line break.)

Part	**LOC**
General logic (ca.uottawa.nudd049.uprove.logic)	1083
Connectives (ca.uottawa.nudd049.uprove.logic.connective)	118
Rules (ca.uottawa.nudd049.uprove.logic.rule)	670
Total logical layer	**1871**
Persistence layer (ca.uottawa.nudd049.uprove.persistence)	163
User interface layer (ca.uottawa.nudd049.uprove.ui)	887
Total production code	**2921**
Test code	3011
Grand total	**5932**

4. Installing

There are several ways to start uProve. One convenient method is to use Java Web Start to launch the application directly from its website, since this method does not require any installation and is totally safe. However, to ensure the latter, the file system is not accessible from within uProve which makes the application unable to save and open proofs. For this to work, the program must be downloaded.

The second simplest method is to download the already compiled program as a jar file. See the website for details on how to do this.

The last method is to download the source code, which is available as a zip file on the website. This section will describe how to install the program from its source file.

The first step is of course to get the zip file (uProve-src.zip) and unpack it. This can be done in several different ways, but on a typical Linux system it can be done as shown below.

```
$ curl -o uProve-src.zip http://uprove.udd.be/download/uProve-src.zip
$ unzip uProve-src.zip
```

The Apache Ant build tool has been used when developing this program and the appropriate build file is provided together with the source files. The tool simplifies the build process but is not required. Sections describing both how to go ahead without Ant and how to do it with Ant are found below.

4.1 Compile and run the program without Ant

An output directory for the compiled class files must be created before continuing.

```
$ mkdir build
```

It is now time to compile uProve. Please note that there should not be any line breaks in this, fairly long, command.

```
$ javac -d build src/ca/uottawa/nudd049/uprove/logic/*.java
src/ca/uottawa/nudd049/uprove/logic/connectives/*.java
src/ca/uottawa/nudd049/uprove/logic/rules/*.java
src/ca/uottawa/nudd049/uprove/persistence/*.java src/ca/uottawa/nudd049/uprove/ui/*.java
```

When this is done it is time to start the application which is done by navigating to the output directory and invoking Java with uProve's full name as its argument.

```
$ cd build
$ java ca.uottawa.nudd049.uprove.logic.Main
```

4.2 Simplify the build process with Ant

Once the source file is unpacked Ant is ready to do its job. As an example the commands for building and running a jar file are shown below.

```
$ ant jar
$ cd build/jar/
$ java -jar uProve.jar
```

Just running the program can be done by invoking the *run* target.

```
$ ant run
```

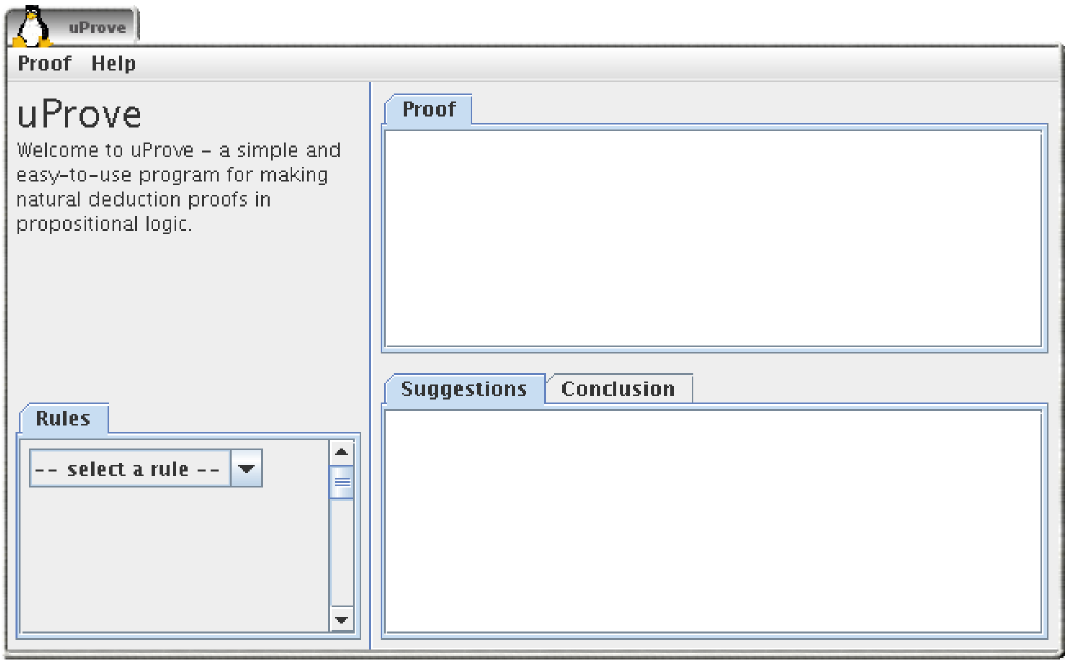
The unit tests can be ran with the help of the *test* target.

```
$ ant test
```

5. Using the program

This section describes how to use the program once it is started. The section presents and explains the user interface. This is probably enough to familiarize the reader with uProve and how to use it. A tutorial is also available on uProve's website.

The exact appearance depends on platform and settings, but the main window should look approximately as the window shown below.

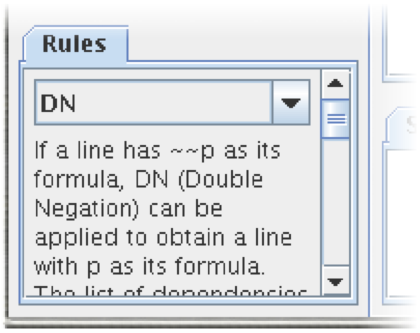


Welcome to uProve - a simple and easy-to-use program for making natural deduction proofs in propositional logic.

The *Proof* menu can be used for starting a new proof, opening and saving a proof and to exit the application. Some help can be shown by using the *Help* menu.

5.1 Rules tab

The *Rules* tab is used for showing information about how a rule may be applied. The rule of interest is selected from the combo box, and a short information text is shown right below it.



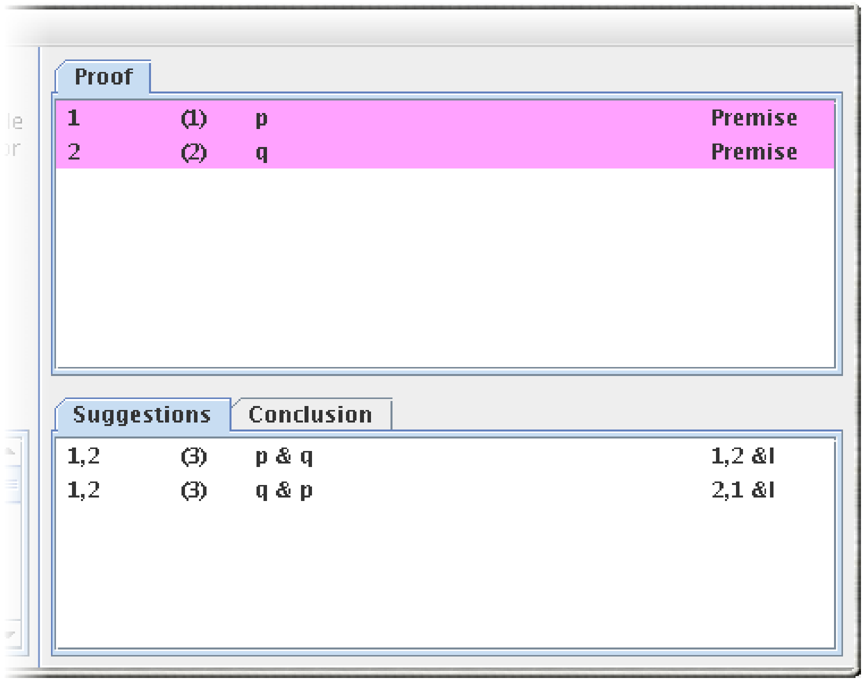
If a line has ~~p as its formula, DN (Double Negation) can be applied to obtain a line with p as its formula.

5.2 Proof tab

The actual proof is showed in the *Proof* tab. Only the premises are shown in this tab when a new proof is started, but as the proof is built more and more lines are added to this panel. When a proof is completed, a text will appear here informing the user about the success.

5.3 Suggestions tab

The proof is being built by, in the proof panel, clicking on the lines to which a rule should be applied. A list of possible rules to apply is then shown in the *Suggestions* tab. The user clicks the appropriate suggestion and the rule is added to the *Proof* tab.



```
1      (1)   p          Premise
2      (2)   q          Premise
```

```
1,2    (3)   p & q      1,2 &I
1,2    (3)   q & p      2,1 &I
```

On the picture above, two lines in the proof are selected and two suggestions are shown. By clicking on one of these two suggested lines that line would be added as a third line in the *Proof* tab.

5.4 Conclusion tab

The *Conclusion* tab contains the wanted conclusion. A user who has forgotten the conclusion s/he is trying to prove can open this tab in order to be reminded.

5.5 Starting a new proof

By selecting to start a new proof, a dialog is shown to the user. That dialog is used to enter premises (if there are any) and a conclusion, which is done by clicking on the corresponding *Add* button. When doing so, a dialog is shown that lets the user enter a formula. Both these dialogs are shown below.

5.6 Entering a formula

In some cases, the result of applying a rule cannot be completely determined by the program. This occurs when the resulting formula contains a placeholder for an arbitrary formula that the user should enter. In such cases a dialog prompting the user to enter a formula will open.

5.7 Opening and saving a proof

The appearance of the dialogs themselves depend on the platform used, but opening and saving proofs is done by selecting the corresponding menu item in the *Proofs* menu. uProve proof files have the extension .upr.

6. Inference rules [1]

The list below presents all inference rules used in uProve together with the name of the respective class in the package of rules.

Assumption (Assumption)

Out:	j	(j)	[?]	Assumption

Copy (Copy)

In:	X	(j)	p	
Out:	X	(k)	p	j Copy

&E (AndElim)

In:	X	(j)	p & q	
Out:	X	(k)	p	j & E
	X	(k)	q	j & E

&I (AndIntro)

In:	X	(j)	p	
	Y	(k)	q	
Out:	X∪Y	(m)	p & q	j,k &I

->E (ImplicationElim)

In:	X	(j)	p->q	
	Y	(k)	p	
Out:	X∪Y	(m)	q	j,k ->E

-> I (ImplicationIntro)

In:	j	(j)	p	Assumption
	X	(k)	q	
Out:	X/j	(m)	p -> q	j,k ->I

~E (NotElim)

In:	X	(j)	~p	
	Y	(k)	p	
Out:	X∪Y	(m)	_|_	j,k -~E

~I (NotIntro)

In:	j	(j)	p	Assumption
	X	(k)	_|_	
Out:	X/j	(m)	~p	j,k ~I

∨E (OrElim)

In:	X	(g)	p ∨ q	
	h	(h)	p	Assumption
	Y	(i)	r	
	j	(j)	q	Assumption
	Z	(k)	r	
Out:	X∪(Y/h)∪(Z/j)	(m)	r	g,h,i,j,k ∨E

∨I (OrIntro)

In:	X	(j)	p	
Out:	X	(k)	p ∨ [?]	j ∨I
	X	(k)	[?] ∨ p	j ∨I

Equivalence Definition 1 (Df1)

In:	X	(j)	(p->q) & (q->p)	
Out:	X	(k)	p <-> q	j Df

Equivalence Definition 2 (Df2)

In:	X	(j)	p <-> q	
Out:	X	(k)	(p->q) & (q->p)	j Df

Double Negation (DN)

In:	X	(j)	~~p	
Out:	X	(k)	p	j DN

Ex Falso Quodlibet (EFQ)

In:	X	(j)	_|_	
Out:	X	(k)	[?]	j EFQ

7. References

[1] Forbes, Graeme (1994). *Modern Logic. A Text in Elementary Symbolic Logic.* New York, USA: Oxford University Press. 397 p. ISBN: 0-19-508028-9 – 0-19-508029-7 (pbk.).