CMPE 58S: Sp. Tp. Computer Aided Verification

Project: Interactive Propositional Logic Engine using Natural Deduction

Mehmet Utkan Gezer

2018700060

December 5, 2018

1 Introduction

Propositional logic, also known as propositional calculus, is the branch of logic dealing with propositions. Natural deduction is a way of handling propositions, whereby we apply a collection of rules to infer new conclusions from zero or more premises, all of which themselves are propositions.

For a more detailed introduction, we will be giving some definitions for propositions and natural deduction, along with its semantics.

1.1 Propositions

Propositions are declarative sentences with a truth value either **true** or **false**, and can be recursively defined as the composition of some other, smaller, propositions. Smallest of propositions are called *atomic* propositions, which are *indecomposable* and are given a unique symbol for the declarations they make.

Compositionals are propositions composed of other propositions, i.e. propositions that are not atomic. In propositional logic, there are 4 different ways of composing a *compositional*:

1. Negation ($\neg$): Negation of a proposition. E.g. "it does not rain" is the negation of "it rains". If the original proposition has the truth value **true**/**false**, then its negation has the truth value **false**/**true**, respectively.

2. Conjunction ($\wedge$): Conjunction of two propositions. E.g. "it does not rain and I am 25" is the conjunction of the propositions "it does not rain" and "I am 25". The compositional has the truth value **true** only if both of its constituents have the truth value **true**, and otherwise **false**.

3. Disjunction ($\vee$): Disjunction of two propositions. E.g. the non-exclusive sense of the statement "it is sunny or I am 5" is the conjunction of the propositions "it is sunny" and "I am 5". The compositional has the truth value **false** only if both of its constituents have the truth value **false**, otherwise (if either one or both of its constituents have the truth value **true**) the compositional is **true**.

4. Implication ($\rightarrow$): Implication of the second proposition, also known as the *consequent*, by the first proposition, also known as the *antecedent*. E.g. "if it is sunny then I am happy" is the implication of the proposition "I am happy" by the proposition "it is sunny". The compositional has the truth value **false** only if the antecedent and consequent are **true** and **false**, respectively.

To formally define the *language* of propositional logic's *well-formed formulas*, we give its Backus Naur form (BNF) as follows:

$$
\begin{aligned}
\langle \phi \rangle \quad &::= \quad \langle \text{atom} \rangle \;\mid\; \Big(\neg\langle\phi\rangle\Big) \;\mid\; \Big(\langle\phi\rangle\wedge\langle\phi\rangle\Big) \;\mid\; \Big(\langle\phi\rangle\vee\langle\phi\rangle\Big) \;\mid\; \Big(\langle\phi\rangle\rightarrow\langle\phi\rangle\Big) \\
\langle \text{atom} \rangle \quad &::= \quad p \;\mid\; q \;\mid\; \ldots \;\mid\; p_1 \;\mid\; p_2 \;\mid\; \ldots
\end{aligned}
$$

While this definition requires each use of a compositional operator to introduce a new pair of parenthesis for the newly generated proposition to be a well-formed formula, in practice we encounter propositons with many of those parenthesis omitted. In absence of parenthesis to enforce an explicit precedence of operator application, following precedence conventions are consulted:

- $\neg$ binds most tightly, followed by $\wedge$, $\vee$, and finally $\rightarrow$, in the given order.

- Implication ($\rightarrow$) is right-associative, i.e. the rightmost implication is to be evaluated the first.

Some books, including *Logic in Computer Science* by Huth and Ryan [1], regard $\wedge$ and $\vee$ operations as equal in precedence, in which case a proposition like $p \vee q \wedge r$ should either be regarded as unintelligible, or the same as $((p \vee q) \wedge r)$. We will be adhering to the above listed convention.

1.2 Natural deduction

Natural deduction is a way of reasoning about a given set of propositions and inferring new ones from them. Using a collection of *proof rules*, natural deduction allows us to come up with *conclusions* starting off by a set of *premises*. This relation between premises and conclusions are formalized by expressions called *sequents*, such as:

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \psi.$$

Sequents with no premises are also valid sequents, and are called *theorems*.

Rules of natural deduction, also known as the previously mentioned proof rules, are at the heart of natural deduction, and this project. They allow us to establish a valid proof step with a proposition, using the previously validated list of propositions.

At any step, we may introduce an *assumption* to the proof, which, however, will introduce an assumption box along with it. The top line of the assumption box is drawn right above the step at which the assumption is introduced. We may close the assumption box after any step. A proof rule may only be applied to propositions, such that;

- Their validity must have been previously established, and
- Either they must be outside any assumption box, or their assumption box must not have been closed, yet.

We refer to such proof steps as the *accessible steps*.

A proof of a sequent is complete when the conclusion of the sequent is established using only the rules of natural deduction and starting off with only the premises of the sequent. It is important to note that an established proposition may not be the conclusion if it is found within an *assumption box*.

Here is a list of all natural deduction rules we have embedded into our program, given in sequents:

Name		**Rule**
Conjunction	Introduction	$\phi, \psi \vdash \phi \land \psi$
	Elimination #1	$\phi \land \psi \vdash \phi$
	Elimination #2	$\phi \land \psi \vdash \psi$
Disjunction	Introduction #1	$\phi \vdash \phi \lor \psi$
	Introduction #2	$\psi \vdash \phi \lor \psi$
	Elimination	$\phi \lor \psi, \boxed{\phi \cdots \chi}, \boxed{\psi \cdots \chi} \vdash \chi$
Implication	Introduction	$\boxed{\phi \cdots \psi} \vdash \phi \implies \psi$
	Elimination	$\phi \implies \psi, \phi \vdash \psi$
Negation	Introduction	$\boxed{\phi \cdots \bot} \vdash \neg\phi$
	Elimination	$\phi, \neg\phi \vdash \bot$
false	Elimination	$\bot \vdash \phi$
Double negation	Introduction	$\phi \vdash \neg\neg\phi$
	Elimination	$\neg\neg\phi \vdash \phi$
MT (Modus Tollens)		$\phi \to \psi, \neg\psi \vdash \neg\phi$
PBC (Proof by Contraditcion)		$\boxed{\neg\phi \cdots \bot} \vdash \phi$
LEM (Law of Excluded Middle)		$\vdash \phi \lor \neg\phi$
Copy		$\phi \vdash \phi$

The boxed premises such as $\boxed{\phi \cdots \psi}$ on implication introduction, is an assumption box in the proof that starts right before the proposition ϕ and ends right after the proposition ψ. To make step on the proof, one of the proof rules given above must be used. To use a rule, we need accessible propositions and/or assumption boxes that fits to the rule's sequent's propositions. Only if we are able to fulfill the *requirements* of the rule, we then may establish the validity of a new proposition according to the definition of the rule, and specifically the rule's sequent's conclusion. In natural deduction, this is the only way to make a proof step and approach to the ultimate conclusion.

Accessibility of an assumption box is defined similar to the accessibility of individual proof steps. This time, not a single step but the assumption box as a whole should be accessible as a single entity.

A proof step is a declaration, and should have a *rationale* stated next to it. Following is a complete list of valid rationales for a proof step:

Rationale	**Can be used next to propositions which...**
Premise:	... are found in the premises of the sequent to be proved.
Proof rule:	... that fit to the conclusion of a proof rule, and only if the corresponding premises of that proof rule is available and validated in a previous step. Those steps must be referenced in the rationale in the same order as they appear in the proof rule.
Assumption:	... appear as the first proof step of an assumption box.

Here is an example of a proof for the sequent $p \vee q, \neg q \vee r \vdash p \vee r$:

1.	$p \vee q$	premise
2.	$\neg q \vee r$	premise
3.	$q \vee \neg q$	*LEM*
4.	q	assumption
5.	$\neg q$	assumption
6.	$\bot$	$\neg_e$ 4, 5
7.	r	$\bot_e$ 6
8.	r	assumption
9.	r	$\vee_e$ 2, 5–7, 8–8
10.	$p \vee r$	$\vee_{i_2}$ 9
11.	$\neg q$	assumption
12.	p	assumption
13.	q	assumption
14.	$\bot$	$\neg_e$ 13, 11
15.	p	$\bot_e$ 14
16.	p	$\vee_e$ 1, 12–12, 13–15
17.	$p \vee r$	$\vee_{i_1}$ 16
18.	$p \vee r$	$\vee_e$ 3, 4–10, 11–17

And here are remarks on this proof:

1. Note that on step #3, we use the rule *LEM* without any argument. It really does not depend on any one of the previous steps, however, it actually has an argument q, which is the reason why it is a rationale for $q \vee \neg q$ and not $\neg r \vee \neg\neg r$.

2. Note the order of arguments on step #14. With respect to the rule of negation elimination ($\neg_e$), second of its two arguments must be the negation-applied of the first one, which is why we first refer to q and then $\neg q$, and not the other way around. Also note that negation-applied is not the same as negation-discarded, although they semantically have the same meaning.

3. Note how assumption boxes are referred to with ranges, i.e. the starting index, followed by a dash, and followed by the ending index.

Hereby we conclude with our definitions on propositional logic and natural deduction, and also our introduction.

2 Interactive Propositional Logic Engine using Natural Deduction

A *propositional logic engine* is an abstract machine that works with propositional logic as its substance. One that works with the methods of natural deduction, is a propositional logic engine *using natural deduction.* With this project, we propose an *interactive* propositional logic engine using natural deduction.

Our product is a software that allows its user to build up valid proofs for any given propositional logic sequent. By stating the rule they want to use and specifying the arguments that want to use it with, users can establish the truth of newer propositions and get them added to the proof as a step. The proof rules are embedded into the software, ensuring the validity of the proof throughout the execution.

The software comes with a command-line interface, which is a REPL[1] that reads user input for proof rules, parses and evaluates it, and re-draws the working proof to the console window, in a loop. The loop ends when the conclusion of the initially input sequent is reached.

In the following subsections, we will be giving out more details on software's workflow, its the input specifications and some of its inner mechanisms. We will also mention some of its software design aspects, some of which have been made possible (with ease) of the programming language Julia.

2.1 Workflow

At all times, the program greets the user with the title same as the name of the project, "Interactive Propositional Logic Engine using Natural Deduction". Initially, the program asks user to provide a sequent that is to be proven.

[1]Read-Eval-Print-Loop

After being provided a sequent, the program enters the proof mode, where it will start displaying a working proof structure with step numbers and box visualizations with ASCII box characters for the assumption boxes. In this mode, the user is repeatedly asked to provide a natural deduction rule to apply. With the provided natural deduction rule, along with its parameters, the software will do either one of the following:

- If the rule is applicable to the given arguments, it will add the proposition, validity of which has been established using the provided proof rule with the given arguments, and then re-draw the proof with the new step on the proof.
- If the rule is malformed:
  - The program may ask the user to re-consider their input, or
  - The program may fail and exit.

The program is designed to apply a properly provided natural deduction rule, and only apply a properly provided natural deduction rule. However, as stated above, it may fail to recover promptly and ask for the user to retry if an input is malformed.

Apart from the natural deduction rules with their parameters, some other keyword statements can be provided to the program to perform the closure of an assumption box, and, for user's convenience, to undo the last action.

When a proof step with the proposition exactly the same as in the conclusion of the initially provided sequent and outside all of the assumption boxes, the program than declares success and exits. This concludes the workflow of the program, and also this subsection.

2.2 Input Specifications

Our program accepts 3 different category of inputs: Propositions, sequents, and natural deduction statements with their arguments. We will provide their specifications separately for a more structured view.

2.2.1 Propositions

Refer to the Subsection 1.1 for the BNF specification for the well-formed formulas on propositions. As stated before in that subsection, we will not expect the user-provided formulas to be well-formed, and tolerantly accept propositions according to precedence rules described in again the same section.

In general, and with respect to the BNF specification, a proposition consists of atoms, negation, conjunction, disjunction, and implication symbols, and finally parenthesis. We extend this list of constituents with constants for tautology and contradiction, which we have seen to take place in propositions, both in the proof rules and the example we gave.

Since we cannot expect users to input Unicode characters like $\wedge$ for the **and** operation, we instead accept sensible ASCII alternatives to represent these operations. We do not accept the Unicode originals, even if the user somehow manages to type them down. Here is the full list of those alternatives, next to their Unicode originals:

Unicode	**Alternative**	
	#1	**#2**
$\top$	`T`	`TRUE`
$\bot$	`F`	`FALSE`
$\neg$	`!`	
$\wedge$	`&`	`*`
$\vee$	`|`	`+`
$\rightarrow$	`->`	
()	`()`	`[]`
q_1	`q1`	

Atomic propositions which are symbolized with a single letter should simply be input using that letter. Atoms with a subscript number should have their number appended right next to them, and not as a subscript. More specifically, all the atoms should be in the form of the following regular expression:

```
[a-z][0-9]*
```

There can be as many spaces around the tokens as the user may input, as well as no space at all. A couple of example propositions our program would accept are as follows:

```
(((!p) & q) -> (p & (q | (!r))))  | (p1 -> (p2 -> (p3 -> p4)))
!p & q -> p & (q | !r)            | p1 -> p2 -> p3 -> p4
!p&q   ->   p&((q)|!r)            | (((p1->p2->p3->p4)))
```

All 6 of the inputs are valid, and the ones on the same column are two different ways of writing the same proposition. The top ones are explicit and well-formed formulas, while the remaining 4 are dependent on conventions of precedence. The bottom ones are untidy examples, where there are a superfluous spaces and then sometimes none. Last examples also exhibit extra parenthesis, which are adequately handled by the parenthesis elimination routine in our program.

2.2.2 Sequents

Refer to the section Introduction and its Subsection 1.2 for the definitions of a sequent. Recall that a sequent with no premises is also a valid sequent, also known as a theorem.

List of premises in a sequent are to be separated with commas (`,`) in our program, just like we do it on paper. The symbol $\vdash$ should be substituted by `=>`. If the user desires to provide a theorem, the sequent symbol `=>` can also be omitted altogether.

Conclusion of the sequent may not be omitted, as it is essential for a sequent. However, if the user desires to use the program for natural deduction without any pre-destined conclusion, he or she may simply provide an unattainable conclusion, such as an atomic proposition `q1234` that does not take place in any one of the premises. This should allow them to run the program indefinitely, as it should be impossible for them to establish the validity of an atomic proposition that does not exist in any one of the premises.

As with the propositions, there can be as many spaces around the tokens as the user feels like, as well as no space at all. A couple of example sequents our program would accept are as follows:

```
!q -> !p => p -> !!q
p -> (q -> r), p, !r => !q
p -> q -> r  , p  , !r=>!q
=> (q -> r) -> ((!q -> !p) -> (p -> r))
(q -> r)   ->  ((!q -> !p) -> (p -> r))
```

Here, second and third, and fourth and fifth sequents are equivalent, i.e. will be interpreted as the same by our program.

2.2.3 Natural deduction statements

Natural deduction rules have no syntax, but they are rather rules to be applied to true propositions in order to establish the truth of newer ones. When a proposition is introduced using a natural deduction rule, however, we have to specify the rule as a rationale next to that proposition in a very specific way.

In our program, we adopt this rationale syntax, as we expect user to provide *natural deduction statements*. In general, a natural deduction statement should start with the identifier of the corresponding natural deduction rule, followed by the list of parameters it takes. Here, parameters are either one of the following three:

- Line number, with the placeholder `#`.
- An interval of line numbers, with the placeholder `#-#`.
- A proposition, with the placeholder `φ`.

The following table specifies all of the natural deduction statements that our program accepts: First two columns are the names of the statements, and corresponds to the names of the natural deduction rules. The next column is the identifier for the statement, and the following columns are placeholders for one of the three different types of parameters as listed above.

The last 3 statements on the table are the only ones that do not have a corresponding natural deduction rule. First two are there to open and close assumption boxes, and the last one is for the convenience of the user, allowing them to undo the effects of the last (non-`undo`) statement they have provided.

Name		**Identifier**	**Arguments**		
			#1	**#2**	**#3**
Conjunction	Introduction	`andi`	`#`		
	Elimination #1	`ande1`	`#`		
	Elimination #2	`ande2`	`#`		
Disjunction	Introduction #1	`ori1`	`#`	φ	
	Introduction #2	`ori1`	`#`	φ	
	Elimination	`ori1`	`#`	`#-#`	`#-#`
Implication	Introduction	`impi`	`#-#`		
	Elimination	`impe`	`#`	`#`	
Negation	Introduction	`negi`	`#-#`		
	Elimination	`nege`	`#`	`#`	
false	Elimination	`bote`	`#`	φ	
Double negation	Introduction	`negnegi`	`#`		
	Elimination	`negnege`	`#`		
MT (Modus Tollens)		`MT`	`#`	`#`	
PBC (Proof by Contraditcion)		`PBC`	`#-#`		
LEM (Law of Excluded Middle)		`LEM`	φ		
Copy		`copy`	`#`		
Assume		`assume`	φ		
Conclude		`conclude`			
Undo		`undo`			

The `#` should be replaced by a single line number out of one of the line numbers that are present in the working proof so far. `#-#` should be replaced by two line numbers separated by a dash (`-`), where the first line number should be of the line right after the start of an assumption box opening line, and the second line number should be of the line right before the end of an assumption box closing line. The φ should be replaced by any proposition, as previously described at the propositions' input specification.

The user may separate the identifier from the first argument either with one or more space characters ( ), or a comma (`,`) with any amount of space characters around it. The arguments should also be separated from each other in a similar fashion.

All the arguments are necessary, and the program will not proceed when they are missing. Program tolerates the natural deduction statement input when the user forgets to provide the necessary φ argument, by letting the user try again. However, forgetting to provide other types of arguments, or providing different types of arguments than necessary, may possibly not be tolerated and result in an error and therefore termination of execution.

We provide the following example natural deduction statements accepted by our program:

```
LEM q
nege 4, 5
nege 4 5
bote 6 p->r
assume (p->n) & (n->p)
conclude
ore 2, 5-7, 8-8
undo
```

Note how we may keep or omit commas as argument/identifier separators. The parser is able to distinguish arguments simply from their forms, and by utilizing the knowledge that no natural deduction statement expects multiple propositons on its arguments.

3 Example execution

4 Conclusion

We are very content with the final version of our program. It is able to produce proofs in a very appealing way, despite being a command-line application. With the interactions of the user, it can prepare any propositional logic proof using natural deduction, having a strict enforcement on its validity at each and every step.

The software has an engine embedded within, and uses that engine at its core to consume input and provide output through a monospace terminal. Overall, the software is intended to be used as an interactive, stand-alone command-line application. However, with only some minor modifications, it can also be turned into a tool and a back-end engine for any other application.

References

[1] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge university press, 2004.