Proof Finder University of Berne

Eveline Lehmann

1 Acknowledgment

This project is an answer to many requests for a replacement of the former LWB-project, which was no longer supported. LWB was a proof-finder framework that was a lot bigger than what you find here. Nevertheless we offer here a framework to find proofs in 4 logics: classical logic, intuitionistic logic and the modal logics K and S4.

It was my final piece of work as an assistant at the institute during my PhD employment. It is the goal to offer a proof finder for some of the most important logics. All the algorithms had been established already and had only to be implemented. The project is based on Python3. We decided to only use standard Python modules to make the tool easy to use. So you should be able to run all programs as soon as you have installed any version of Python3. To have access on the proofs you must install LaTeXtoo.

As I am still a beginner in Python, there are a lot of things that can be improved. During the process I learned a lot about how to use Python and it's spacial features. Some of them are implemented, others are not. I hope that in a later version at least some parts will be improved. I'm aware of many small things that I like to improve as soon as possible. If you see things that should be improved with a certain urgency, please contact the LTG-group of the Institute of Computer Science at the University of Berne.

I would like to thank Prof. Dr. Thomas Studer and Dr. Peppo Bramnilla for their support and feedback for this project.

2 Modules and Files

You find several modules and files which can be grouped as follows:

- The GUI module to interact in an intuitive way and the parsing module which allows to enter formulas in an easy way. Details can be found in Section 2.1.
- The 4 logical modules providing a derivability check and where a prooftree is built. How to use them without GUI is described in Section 2.2.
- The files to create proofTrees and export them to LaTeX-files. You might never use them by your own. Nevertheless you find more information about them in Section 2.3.
- A directory of test-files and a module to create pigeon-hole formulas. You find a description of them in Section 2.4.

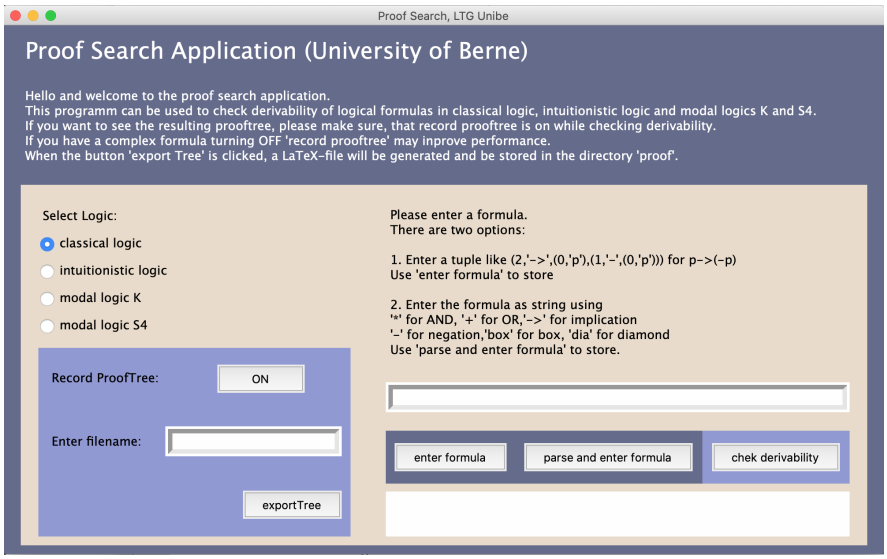

Figure 1: The GUI for testing single formulas.

2.1 The module gui.py

The easiest way to interact with the proof-finder modules is by the *gui.py* file. It contains a GUI with everything you need to test the derivability of single formulas (see Figure 1).

On the left handside you can chose the logic. You can chose between

- classical logic, where the derivability is implemented with use check according to Peppo Brambilla, Proof Search in Propositional Circumscription Logic, PhD thesis University of Bern, p33-36.
- intuitionistic logic, where derivability is implemented with GHPC according to Dickhoff Roy, Contraction-free Sequent Calculi for Intuitionistic Logic, in: The Journal of Symbolic Logic, Volume 57, 1992, p.795-807, here p.801.
- modal logics K and S4, where derivability is implemented with scK and S4 according to Hererding Alain, Sequent Calculi for Proof Search in Some Modal Logics, PhD thesis University of Bern, Definitions 5.2.2 and 5.6.15.

On the right handside you have an entry field in which you can write your formula. You can chose between enter a tuple that represents a formula and using standard symbols to enter the formula as a string. The first way is analogous to the description in Section 2.2. Please see there how define formulas in this case. To enter a formula in its tuple-representation use the button "enter formula".

We expect that most people prefer a string representation of the formula. This is possible by replacing $\neg$ with -, $\vee$ with +, $\wedge$ with $*$, $\rightarrow$ with ->, $\Box$ with 'box' and $\Diamond$ with 'dia'.

You can name your atoms as you like as long as you do not use any keywords which are: (,), -, ->, +, $*$, box, dia. Then press "parse and enter formula" and if sucessful you can check derivability with the button "check derivability".

Here are some examples how to enter formulas:

formula	enter the following
$\neg p$	- p
$p \wedge q$	p * q
$p \vee q$	p + q
$p \rightarrow q$	p ->q
$\Box p$	box p
$\Diamond p$	dia p
$\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$	box (p ->q) ->(box p ->box q)

Brackets are needed except for separating unary operators from one another. You may add brackets where they are not necessary. Superfluous brackets are ignored. The module parsingString will automatically transform the formulas into tuples as soon as you press the button "parse and enter formula".

The formula you see in the entry field is always the current formula. So if you change the formula in the entry field, you have no longer access to the previous formula regardless whether you haven't entered the new formula yet.

If the field "Record ProofTree" is on, the process may take a bit longer because a tree representation of the proof will be generated during the derivability process. If ProofTree was on, you can export this tree to a LaTeX-file. You can chose the name of your file. The LaTeX-file will then be saved in the subdirectory *proof*. You then can extract the file to PDF by the new buttom just below "export Tree". If the command pdfLatex is not available in your operating system, this will not work and you have to compile the LaTeX-file manually.

Please note, that for complex formulas and long prooftrees the space of a normal PDF-page may not be enough. So you may have to copy the LaTeX-code and fragment it into smaller pieces. In the LaTeX-file we use the package bussproof to illustrate the prooftree.

2.2 Derivability: cpc_uc.py, ghpc_backtracking_first.py, ghpc_branching_first.py, s4.py, scK.py

For each logic CL, IL, K and S4 there is a program with the corresponding derivability funkction.

- The derivability check for **classical logic** is in file cpc_uc.py, where the derivability is implemented with use check according to Peppo Brambilla, Proof Search in Propositional Circumscription Logic, PhD thesis University of Bern, p33-36.
- For **intuitionistic logic** there are two implementations that differ, which rules are applied first: ghpc_backtracking_first.py, ghpc_branching_first.py. Derivability is implemented with GHPC sequent calculus according to Dickhoff Roy, Contraction-free Sequent Calculi for Intuitionistic Logic, in: The Journal of Symbolic Logic, Volume 57, 1992, p.795-807, here p.801. You can choose whether you want to do backtracking rules first or branching ones. As far as we know it only depends on the formula which way is faster.
- The others are for **modal logics K** (file scK.py) and **S4** (file s4.py), where derivability is implemented with scK and S4 according to Hererding

Alain, Sequent Calculi for Proof Search in Some Modal Logics, PhD thesis University of Bern, Definitions 5.2.2 and 5.6.15.

In all cases to check derivability you can use the function *derivable(formula)*, where formula should be a tuple built according to the rules explained below. If you want to have the proof recorded into a tree, you should instead call *derivable(formula, True)*. In this case a prooftree will be built during the derivation process. Having added this argument you can use the function to_file() from module graph to generate a LaTeX-file containing the proof. For more details consult Section 2.3.

The form of well-defined formulas depends on the language of the logic of course. But the general pattern is as follows:
Formulas must be given as a tuple of

- 2 (in case of an atom)
- 3 (in case of a negation or a modal)
- 4 (in case of a binary operator)

elements, such that

- at index 0 is the arity of the operation:
  - 0 if atom;
  - 1 if $\neg, \Box, \Diamond$;
  - 2 if $\wedge, \vee, \rightarrow$.
- at index 1: the operation symbol '-' (for $\neg$), '+' (for $\vee$), '$*$' (for $\wedge$), '->' (for $\rightarrow$), 'box' (for $\Box$), 'dia' (for $\Diamond$) or a string in case of an atom.
- at index 2 (if no atom): another tuple according to these rules, representing the first argument of the operation.
- at index 3 (if it is a binary operation): another tuple according to these rules, representing the second argument of the operation.

Here are some examples how to transform formulas in tuples:

formula	enter the following
p	(0,'p')
$\neg p$	(1,'-',(0, 'p')
$p_0 \wedge p_1$	(2,'$*$',(0,'p0'),(0,'p1'))
$p \vee q$	(2,'+',(0,'p'),(0, 'q'))
$p \rightarrow q$	(2,'->',(0,'p'),(0,'q'))
$\Box p$	(1,'box',(0,'p'))
$\Diamond p$	(1,'dia',(0,'p'))
$\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$	(2,'->',(1,'box',(2,'->',(0,'p'),(0,'q'))), (2,'->' (1,'box',(0,'p')),(1,'box',(0,'q'))))

2.3 Prooftree and LaTeX-files

The relevant files here are: graph.py, text_intro, text_end.

The main file here is graph.py. You find three implemented classes: sequents, nodes and prooftrees.

Sequents have as attributes a list of two lists and a string representation. The string representation transforms a list of two tuples representing each a formula according to Section 2.2 into a LaTeX-representation of a sequent containing these formulas.

Nodes contain a sequent, a rule (string), a parent node and a list of children (nodes). It is able to add an remove children.

A prooftree has a root and is able to generates a LaTeX-file. To generate this LaTeX-file it uses the file text_intro for the preamble and text_end for the final lines.

If you derive a formula either in the GUI or by running the logical modules, you can record the steps of the derivation process in a prooftree. In the GUI you just check that "Record ProofTree" is on. If you run a logical module use the function *derivable(formula)* with a second argument True, i.e. *derivable(formula, True)*. This generates that a prooftree is built. To print it in a LaTeX-file use the function to_file() of the graph module with two optional arguments *filename* and *directory*. This generates a LaTeX-file that entails the source code for a prooftree using the bussproof package. If you leave away the arguments, the file will be named *proof_to_picture.tex* and be stored in a directory named *proof* in your working directory.

2.4 Tests

In the subdirectory *tests* you find tests for most of the modules. Feel free to add more.

There is also a module *pigeonHole.py* that can be used to generate formulas for the pigeon hole principle. You can generate them with *generate_ph(n, True)* or *generate_ph(n, False)*, where n denotes the number of pigeons. The boolean value determines whether you have n holes and hence the principle fails in case of False or, in case of True, $n-1$ holes and therefore there is at least one hole with more than one pigeon in it.