Constructive Logic (15-317), Fall 2017
Assignment 5: Computing proofs

Contact: Daniel Gratzer (jozefg@cmu.edu)

Due Thursday, October 19, 2017, 1:30pm

This assignment is due at the beginning of class on the above date and must be submitted electronically via autolab. Submit your homework as a single file, **not a tar**. Just submit **hw5.sml**. **After submitting via autolab, please check the submission's contents to ensure it contains what you expect. No points can be given to a submission that isn't there.**

You might have noticed, after some practice, that proving a theorem in a calculus becomes quite a mechanical task. Wouldn't it be great if we could have the computer do that for us? That is exactly our goal for this homework: to implement an automatic theorem prover for propositional intuitionistic logic.

The first thing to think about is which calculus we will use. It should be clear by now that natural deduction is not the best choice, as it is too non-deterministic. The verification calculus could be a bit better, as it avoids the redundant steps that eliminate and introduce the same connective over and over again, but it still has the problem of keeping track of the right assumptions at the right places. Maybe we should try the sequent-style presentation of natural deduction, since this keeps the context in place. In this case we need to be very smart about which direction to work at each step, since we can either go upwards or downwards. Instead of trying to come up with heuristics for that, why don't we use the sequent calculus itself, where proof construction always happens from the bottom up?

Indeed, sequent calculi are much better behaved for proof search. But we need to be careful about it. Think about the first sequent calculus we have seen. In this first version, the formulas on the left side of the sequent were persistent. This means we can *always* choose to decompose those formulas. In fact, any sequent calculus that has what we call *implicit contraction*[1] of some formulas runs into the same problem. Roy Dyckhoff's contraction-free sequent calculus avoids these problems. This calculus, called **G4ip**, relies on distinguishing the type of

[1] Usually in the form of applying a rule to decompose a formula and keeping a copy of the original formula in the context.

antecedent on an implication on the left. For reference, the rules are below. For further information, please see the notes for Lecture 11.

Identity

$$\frac{P \in \Gamma}{\Gamma \longrightarrow P}\ \mathsf{id}_P$$

Ordinary Rules

$$\frac{}{\Gamma \longrightarrow \top}\ \top R \qquad \frac{\Gamma \longrightarrow C}{\Gamma, \top \longrightarrow C}\ \top L$$

$$\frac{\Gamma \longrightarrow A \quad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B}\ \wedge R \qquad \frac{\Gamma, A, B \longrightarrow C}{\Gamma, A \wedge B \longrightarrow C}\ \wedge L$$

(no $\bot R$ rule) $\qquad \dfrac{}{\Gamma, \bot \longrightarrow C}\ \bot L$

$$\frac{\Gamma \longrightarrow A}{\Gamma \longrightarrow A \vee B}\ \vee R_1 \quad \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow A \vee B}\ \vee R_2 \qquad \frac{\Gamma, A \longrightarrow C \quad \Gamma, B \longrightarrow C}{\Gamma, A \vee B \longrightarrow C}\ \vee L$$

$$\frac{\Gamma, A \longrightarrow B}{\Gamma \longrightarrow A \supset B}\ \supset R$$

Compound Left Rules

$$\frac{P \in \Gamma \quad \Gamma, B \longrightarrow C}{\Gamma, P \supset B \longrightarrow C}\ P \supset L$$

$$\frac{\Gamma, B \longrightarrow C}{\Gamma, \top \supset B \longrightarrow C}\ \top \supset L \qquad \frac{\Gamma, A_1 \supset (A_2 \supset B) \longrightarrow C}{\Gamma, (A_1 \wedge A_2) \supset B \longrightarrow C}\ \wedge \supset L$$

$$\frac{\Gamma \longrightarrow C}{\Gamma, \bot \supset B \longrightarrow C}\ \bot \supset L \qquad \frac{\Gamma, A_1 \supset B, A_2 \supset B \longrightarrow C}{\Gamma, A_1 \vee A_2 \supset B \longrightarrow C}\ \vee \supset L$$

$$\frac{\Gamma, A_2 \supset B \longrightarrow A_1 \supset A_2 \quad \Gamma, B \longrightarrow C}{\Gamma, (A_1 \supset A_2) \supset B \longrightarrow C}\ \supset \supset L$$

Because **G4ip**'s rules all reduce the "weight" of the formulas making up the sequent when read bottom-up, it is straightforward to see that it represents a decision procedure. The rules themselves are non-deterministic, though, so one must invest some effort in extracting a deterministic implementation from them.

Task 1 (40 pts)**.** Implement a proof search procedure based on the **G4ip** calculus. Efficiency should not be a primary concern, but see the hints below regarding invertible rules. Strive instead for *correctness* and *elegance*, in that order.

You should write your implementation in Standard ML.[2] Some starter code is provided in the file `prop.sml`, included in this homework's handout, to clarify the setup of the problem and give you some basic tools for debugging Implement a structure `G4ip` matching the signature `G4IP`. A simple test harness assuming this structure is given in the structure `Test` in the file `test.sml`, also included in the handout. Feel free to post any additional interesting test cases you encounter to Piazza.

Here are some hints to help guide your implementation:

- Be sure to apply all invertible rules before you apply any non-invertible rules. Recall that the non-invertible rules in **G4ip** are id_P, $\vee R_1$, $\vee R_2$, $\supset\supset L$, and $P\supset L$. Among these, id and $P\supset L$ have somewhat special status: if they apply, we don't need to look back because there is no premise (id_P), or the sequent in the premise is provable whenever the conclusion is ($P\supset L$).

  One simple way to ensure that you do inversions first is to maintain a second context of non-invertible propositions and to process it only when the invertible rules have been exhausted.

- When it comes time to perform non-invertible search, you'll have to consider all possible choices you might make. Many theorems require you to use your non-invertible hypotheses in a particular order, and unless you try all possible orders, you may miss a proof.

- The provided test cases can help you catch many easy-to-make errors. Test your code early and often! If you come up with any interesting test cases of your own that help you catch other errors, we encourage you to share them on Piazza.

There are many subtleties and design decisions involved in this task, so don't leave it until the last minute!

[2] If you are not comfortable writing in Standard ML, you should contact the instructors and the TAs for help.