Constructive Logic (15-317), Spring 2021
Assignment 8: Theorem Proving and Prolog

Instructor: Karl Crary
TAs: Avery Cowan, Katherine Cordwell, Matias Scharager, Antian Wang

Submit to Gradescope by Wednesday, November 3, 2021, 11:59 pm

For this homework, you will be submitting both SML and Prolog to Gradescope:

- `hw.sml` (your SML solution to theorem proving) should go to Homework 8 (SML).
- `hw.pl` (your Prolog solution to graph coloring) should go to Homework 8 (Prolog).

Be sure to submit to both assignments.

1 Implementing a theorem prover

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. The holy grail of constructive logic would be a prover that could automatically generate a natural deduction proof for an arbitrary proposition. However, it should be clear by now that natural deduction is not the best choice for search, 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 context-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. The inversion calculus avoids these problems. This calculus refines the restricted sequent calculus into two mutually dependent forms of sequents.

$$
\begin{array}{ll}
\Delta^- \,;\, \Omega \xrightarrow{R} C & \text{Decompose } C \text{ on the right} \\
\Delta^- \,;\, \Omega \xrightarrow{L} C^+ & \text{Decompose } \Omega \text{ on the left}
\end{array}
$$

Above, Ω is a plain context containing any kind of formula. Δ^- is a context restricted to those formulas whose left rules are *not* invertible, and C^+ in the second sequent is a formula whose right rule is *not*

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

invertible. Both types of sequents can also contain atoms. All the rules of the inversion calculus have the property that the "weight" of the sequents decrease when the rules are read bottom-up, *except* the left rule for implication. Use of the implication left rule can lead to looping because we can continue to try using the left implication rule on the same implication in the premise. As such, we will be extending inversion to use Roy Dyckhoff's contraction-free sequent calculus. His calculus, called **g4ip**, relies on distinguishing the type of antecedent on an implication on the left. To perform efficient proof search, we are extending the inversion calculus with a new form of judgment to apply synchronous (non-invertible) rules; these include the right rules of right-synchronous connectives and the left rules of left-synchronous connectives.

For further information, please see the notes posted along with the homework. They have an excellent description of combining **g4ip** with the inversion calculus, along with suggestions for implementing these rules as a decision procedure in a functional programming language.

Our forms of judgment are as follows:

$$\begin{array}{ll} \Delta^-; \Omega \xrightarrow{\text{g4ip}}_{\mathsf{R}} C & \text{Decompose } C \text{ on the right} \\ \Delta^-; \Omega \xrightarrow{\text{g4ip}}_{\mathsf{L}} C^+ & \text{Decompose } \Omega \text{ on the left} \\ \Delta^- \xrightarrow{\text{g4ip}}_{\mathsf{S}} C^+ & \text{Apply non-invertible rules} \end{array}$$

The rules of our version of **g4ip** are divided roughly into the inversion phases (right and left) and the search phase. Unlike in the inversion calculus, we have the search rule to make a formal distinction between $\Delta^-; \cdot \xrightarrow{\text{g4ip}}_{\mathsf{L}} C^+$ and $\Delta^- \xrightarrow{\text{g4ip}}_{\mathsf{S}} C^+$. Purposefully, the search phase does not have the secondary Ω context anymore, because search should only happen once all left-invertible propositions in Ω are processed and all left-noninvertible propositions in Ω are shifted into Δ. For clarity, we can formally define the polarities mentioned in the rules. Positive and negative correspond exactly with right-synchronous propositions and left-synchronous propositions.

$$\begin{array}{l} C^+ ::= A \vee B \mid \bot \mid P \\ C^- ::= (A_1 \supset A_2) \supset B \mid P \supset B \mid P \end{array}$$

Above, P is considered atomic while all other metavariables are arbitrary propositions. Based on these polarizations, the negative context Δ^- is composed entirely of negative propositions.

For reference, the rules for **g4ip** are in Figure 1. 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 first part is a repeat from last week. You should include it in your solution, and the grader will compile and test it, but those tests will not affect your score. (The purpose for this is to help you find bugs in your implementation.)

Task 1 (0 points)**.** Implement a proof search procedure based on the **inversion extended g4ip**. This algorithm should take in a right inversion judgement and produce an inversion with **g4ip** proof option. Efficiency should not be a primary concern, but see the hints below regarding invertible rules. Strive instead for *correctness* and *elegance*, in that order.

To compile your code locally run `smlnj -m sources.cm`. To activate pretty printing during local testing call `Printing.activate()` from your repl.

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 init, $\vee\mathsf{R}_1$, $\vee\mathsf{R}_2$, $P{\supset}\mathsf{L}$ and ${\supset}{\supset}\mathsf{L}$. Among these, init and $P{\supset}\mathsf{L}$ have somewhat special status: if they apply, we don't need to look back because there is no premise (init), or the sequent in the premise is provable whenever the conclusion is ($P{\supset}\mathsf{L}$).

Right Inversion

$$\frac{\Delta^-;\Omega \xrightarrow{\mathsf{g4ip}}_{\mathsf{R}} A \quad \Delta^-;\Omega \xrightarrow{\mathsf{g4ip}}_{\mathsf{R}} B}{\Delta^-;\Omega \xrightarrow{\mathsf{g4ip}}_{\mathsf{R}} A \wedge B}\ \wedge\mathsf{R} \qquad \frac{\Delta^-;\Omega, A \xrightarrow{\mathsf{g4ip}}_{\mathsf{R}} B}{\Delta^-;\Omega \xrightarrow{\mathsf{g4ip}}_{\mathsf{R}} A \supset B}\ \supset\mathsf{R} \qquad \frac{}{\Delta^-;\Omega \xrightarrow{\mathsf{g4ip}}_{\mathsf{R}} \top}\ \top\mathsf{R}$$

Switching Mode

$$\frac{\Delta^-;\Omega \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}{\Delta^-;\Omega \xrightarrow{\mathsf{g4ip}}_{\mathsf{R}} C^+}\ \mathsf{LR}_+$$

Left Inversion

$$\frac{\Delta^-;\Omega, A, B \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}{\Delta^-;\Omega, A \wedge B \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}\ \wedge\mathsf{L} \qquad \frac{\Delta^-;\Omega, A \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+ \quad \Delta^-;\Omega, B \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}{\Delta^-;\Omega, A \vee B \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}\ \vee\mathsf{L} \qquad \frac{}{\Delta^-;\Omega, \bot \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}\ \bot\mathsf{L}$$

$$\frac{\Delta^-;\Omega \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}{\Delta^-;\Omega, \top \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}\ \top\mathsf{L}$$

Compound Left Invertible Rules

$$\frac{\Delta^-;\Omega, B \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}{\Delta^-;\Omega, \top \supset B \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}\ \top{\supset}\mathsf{L} \qquad \frac{\Delta^-;\Omega, A_1 \supset A_2 \supset B \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}{\Delta^-;\Omega, (A_1 \wedge A_2) \supset B \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}\ \wedge{\supset}\mathsf{L}$$

$$\frac{\Delta^-;\Omega, A_1 \supset B, A_2 \supset B \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}{\Delta^-;\Omega, (A_1 \vee A_2) \supset B \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}\ \vee{\supset}\mathsf{L} \qquad \frac{\Delta^-;\Omega \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}{\Delta^-;\Omega, \bot \supset B \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}\ \bot{\supset}\mathsf{L}$$

Shift and Search

$$\frac{\Delta^-, A^-;\Omega \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}{\Delta^-;\Omega, A^- \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}\ \mathsf{shift} \qquad \frac{\Delta^- \xrightarrow{\mathsf{g4ip}}_{\mathsf{S}} C^+}{\Delta^-;\cdot \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}\ \mathsf{search}$$

Search Rules

$$\frac{P \in \Delta^-}{\Delta^- \xrightarrow{\mathsf{g4ip}}_{\mathsf{S}} P}\ \mathsf{init} \qquad \frac{\Delta^-;\cdot \xrightarrow{\mathsf{g4ip}}_{\mathsf{R}} A}{\Delta^- \xrightarrow{\mathsf{g4ip}}_{\mathsf{S}} A \vee B}\ \vee\mathsf{R}_1 \qquad \frac{\Delta^-;\cdot \xrightarrow{\mathsf{g4ip}}_{\mathsf{R}} B}{\Delta^- \xrightarrow{\mathsf{g4ip}}_{\mathsf{S}} A \vee B}\ \vee\mathsf{R}_2$$

Compound Left Search Rules

$$\frac{P \in \Delta^- \quad \Delta^-; B \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}{\Delta^-, P \supset B \xrightarrow{\mathsf{g4ip}}_{\mathsf{S}} C^+}\ P{\supset}\mathsf{L} \qquad \frac{\Delta^-; A_2 \supset B, A_1 \xrightarrow{\mathsf{g4ip}}_{\mathsf{R}} A_2 \quad \Delta^-; B \xrightarrow{\mathsf{g4ip}}_{\mathsf{L}} C^+}{\Delta^-, (A_1 \supset A_2) \supset B \xrightarrow{\mathsf{g4ip}}_{\mathsf{S}} C^+}\ {\supset}{\supset}\mathsf{L}$$

Figure 1: G4ip

- 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.

- Since you will be creating your own proof trees algorithmically, make sure to read the `inversion_g4ip.sml` file carefully. We also won't have the usual macros to create a proof tree so looking at the definition of proofs in `logic.sml` might also be helpful.

- Generating a proof can be difficult and create incredibly nested cases. As such, we suggest using continuation passing style (CPS) to simplify your code. For a refresher on CPS, please feel free to look at the 15-150 notes on the topic.

- Your code should be elegant and use good style. Please look at the style guide from the pre-requisite course 15-150 to guide your stylistic decisions in SML.

- The rules themselves are non-deterministic, so one must invest some effort in extracting a deterministic implementation from them. Planning before coding will likely save time.

- Test your code early and often! If you come up with any interesting test cases that help you catch 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!

Task 2 (25 points). We now have a prover that generates a proof in inversion calculus with **g4ip**. But our holy grail is a generic natural deduction prover. For this task, we will bridge this gap and use the inversion calculus process to create a natural deduction proof.

Implement a function that takes in a natural deduction proposition and returns a natural deduction proof option. There are two common approaches to this problem. The first essentially replicates your code from task 1 but constructs the natural deduction proof instead of the inversion proof when combining the premises. The second approach is to write a function that can translate a **g4ip** proof to a natural deduction proof. Then you can simply use your prover from task 1 and translate the answer to natural deduction. A third approach is to instrument your prover to generate proof terms, and then write code to reconstruct a natural-deduction typing derivation from a proof term. Feel free to use either approach or to design your own algorithm.

To compile the full code for final submission run `smlnj -m sources.cm` as always.

Here are some hints to help guide your implementation:

- The hardest part of this task will likely be dealing with left rules since elimination rules work downward while left rules work upward. One idea might include having your translator/prover function keep track of a justification for items in the context.

- We have provided the function `Fresh.fresh()` which produces a unique string every time you call it for easier generation of hypothesis names.

- The normal natural deduction proof tree macros will not be provided in the starter code. Therefore, taking a second look at `natural_deduction.sml` and `logic.sml` might prove helpful.

- As before, feel free to share any interesting test cases on Piazza.

Setting up Prolog

1. SSH into unix.andrew.cmu.edu
2. Execute this command: `/afs/andrew/course/15/317/bin/317setup`
3. Run Prolog with the `prolog` command (exit with control-D or "`halt.`")

2 Coloring maps

Graph coloring is an interesting problem in graph theory. A graph coloring is an assignment of colors to each vertex such that no two adjacent vertices have the same color. Of particular interest is a coloring using a minimum number of colors; this number is called the *chromatic number* of the graph. The four-color theorem states that any planar graph[2] can be colored using at most four colors. The theorem was proved in 1976 using a computer program, and has caused much controversy (is a computer proof really a proof?). It has since been formally verified using the Coq theorem prover in 2005.

As a consequence of this theorem, any map can be colored with at most four colors such that no adjacent regions have the same color. This is because every map can be represented by a planar graph, with one vertex for each region, and an edge between two vertices if and only if their corresponding regions are adjacent.

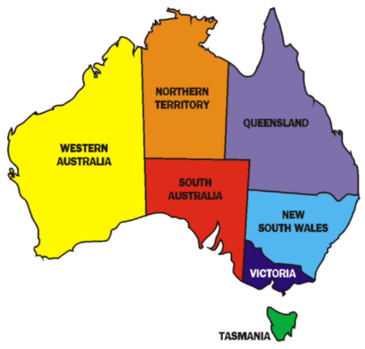

Figure 2: Australia (more colorful than necessary)

Consider, for example, Australia's map in Figure 2. Observe that this map uses more colors than necessary, although this might make it more visually appealing.

Task 3 (15 points)**.** Implement a predicate `color_graph`$(nodes, edges, colors)$ that associates with the graph $(nodes, edges)$ all of the valid 4-colorings of the graph. Submit your implementation in a file named `hw.pl`.

The predicate `color_graph` should find all valid colorings via backtracking. For efficiency reasons, you may prefer to find all valid colorings without repetition, but we will not be checking this. Once all valid solutions have been found via backtracking, the predicate should fail. You may assume the graph is finite, and your implementation should satisfy the following requirements:

1. You should define a `color/1` predicate with four colors.
2. Assume there are predicates `node/1` and `edge/2` that each take in atoms.

[2]A graph that can be drawn on the plane with no crossing edges.

3. In `color_graph/3`, the first parameter is a list of `node/1` terms, the second parameter is a list of `edge/2` terms, and the third parameter is a list of pairs `(a,c)`, which indicates that `node(a)` is colored with `color(c)`.

4. The predicate `color_graph` should have mode `color_graph`$(+nodes, +edges, -coloring)$.

5. Given ground inputs for the first two arguments (*i.e.,* nodes and edges), `color_graph` should always return at least one coloring. (You will not be given an graph that cannot be 4-colored.) If Prolog is asked to backtrack and generate additional solutions, `color_graph` should return all possible colorings.

Your solution does not need to be very long. The reference solution is just 19 lines of Prolog, including the color definitions.

For your convenience, we have provided you with a shell script to test your implementation. You can invoke it by executing:

```
$ ./test_coloring.sh
```

The test script uses another Prolog file (`coloring_tests.pl`) to test your implementation. It employs some Prolog features that you have not been taught. We strongly urge you not to use any of those features in your solution. They are unlikely to help, and are very likely to make your code hard to understand.