Constructive Logic (15-317), Fall 2019
Assignment 8: Computing proofs

Instructor: Karl Crary
TAs: Avery Cowan, David Kahn, Siva Somayyajula

Due: Friday, October 25, 2019, 11:59 pm

There is no written component this week. Submit your programming homework as a single `inversion_calculus.sml` file to Autolab. **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.**

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. 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. The inversion calculus avoids these problems. This calculus refines the restricted sequent calculus into two mutually dependent forms of sequents.

$$\Delta^- \,;\, \Omega \xrightarrow{R} C \qquad \text{Decompose } C \text{ on the right}$$
$$\Delta^- \,;\, \Omega \xrightarrow{L} C^+ \qquad \text{Decompose } \Omega \text{ on the left}$$

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* invertible. Both types of sequents can also contain atoms. For reference, the rules are below. For further information, please see the notes for Lecture 12.

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

Right Invertible Rules

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

Switching Modes

$$\frac{\Delta^- \,;\, \Omega \xrightarrow{L} P}{\Delta^- \,;\, \Omega \xrightarrow{R} P}\ \mathsf{LR}_P (P \text{ atomic}) \qquad \frac{\Delta^- \,;\, \Omega \xrightarrow{L} A \vee B}{\Delta^- \,;\, \Omega \xrightarrow{R} A \vee B}\ \mathsf{LR}_\vee \qquad \frac{\Delta^- \,;\, \Omega \xrightarrow{L} \bot}{\Delta^- \,;\, \Omega \xrightarrow{R} \bot}\ \mathsf{LR}_\bot$$

Left Invertible Rules

$$\frac{\Delta^- \,;\, \Omega \cdot A \cdot B \xrightarrow{L} C^+}{\Delta^- \,;\, \Omega \cdot (A \wedge B) \xrightarrow{L} C^+}\ \wedge L \qquad \frac{\Delta^- \,;\, \Omega \xrightarrow{L} C^+}{\Delta^- \,;\, \Omega \cdot \top \xrightarrow{L} C^+}\ \top L \qquad \frac{}{\Delta^- \,;\, \Omega \cdot \bot \xrightarrow{L} C^+}\ \bot L$$

$$\frac{\Delta^- \,;\, \Omega \cdot A \xrightarrow{L} C^+ \quad \Delta^- \,;\, \Omega \cdot B \xrightarrow{L} C^+}{\Delta^- \,;\, \Omega \cdot (A \vee B) \xrightarrow{L} C^+}\ \vee L$$

Shift Rules

$$\frac{\Delta^-, P \,;\, \Omega \xrightarrow{L} C^+}{\Delta^- \,;\, \Omega \cdot P \xrightarrow{L} C^+}\ \mathsf{shift}_P\ (P \text{ atomic})$$

Search Rules

$$\frac{P \in \Delta^-}{\Delta^- \,;\, \cdot \xrightarrow{L} P}\ \mathsf{init}\ (P \text{ atomic}) \qquad \frac{\Delta^- \,;\, \cdot \xrightarrow{R} A}{\Delta^- \,;\, \cdot \xrightarrow{L} A \vee B}\ \vee R_1 \qquad \frac{\Delta^- \,;\, \cdot \xrightarrow{R} B}{\Delta^- \,;\, \cdot \xrightarrow{L} A \vee B}\ \vee R_2$$

The one true answer for resolving implication in proof search

In class, we saw a version of the **inversion calculus** for intuitionistic propositional logic (with truth, falsehood, disjunction, conjunction and implication); 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. If it weren't for this rule, we could therefore use the inversion calculus directly to implement a terminating decision procedure for intuitionistic propositional logic.

There are two standard solutions to this problem in the literature: the hacker's solution is to implement *loop detection* in order to ensure that the proof search process always bottoms out; a more elegant solution, which we have already seen, is provided by Dyckhoff's *contraction-free sequent calculus*, called **g4ip**, which decomposes the $\supset L$ rule into several different rules.

In this assignment, we will choose neither, and instead do something far more effective. We will wave our hands and shake our heads insisting that implication has been a hoax all along. Thus, by not allowing any rules or propositions that use the implication connective we have resolved the issue of implication entirely, allowing us to implement a proof search engine for the *implication-free fragment* of the inversion calculus, where we omit the implication connective. This works because implication is isn't real and doesn't exist so stop bringing it up. Next week we'll consider implication—and how to use **g4ip** to include it in proof search.

Task 1. Implement a proof search procedure based on the **inversion calculus** without implication. 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]

Tip The rules themselves are non-deterministic, so one must invest some effort in extracting a deterministic implementation from them.
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.

```
signature PROP =
sig
  datatype prop =                 (* A ::=            *)
      Atom of string              (*       P          *)
    | True                        (*     | T          *)
    | And of prop * prop          (*     | A1 & A2    *)
    | False                       (*     | F          *)
    | Or of prop * prop           (*     | A1 | A2    *)

  val toString : prop -> string
  val eq : prop * prop -> bool
end

structure Prop :> PROP
```

Your task is implement a structure `InversionCalculus` matching the signature `INVERSION_CALCULUS`. The signature has been provided below, and is included in the handout materials.

```
signature INVERSION_CALCULUS =
sig
  (* decide D A = true   iff  . ; D --R--> A has a proof,
     decide D A = false  iff  . ; D --R--> A has no proof
   *)
  val decide : Prop.prop list -> Prop.prop -> bool
end
```

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

Testing

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 **inversion calculus** are init, $\vee R_1$, $\vee R_2$.
- When it comes time to perform non-invertible search, you'll have to consider all possible choices you might make.
- 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!