# Constructive Logic (15-317), Fall 2019
# Assignment 6: Propositional Theorem Proving

Instructor: Andr Platzer
TAs: Avery Cowan, Carter Williams, Cam Wong, Klaas Pruiksma

Due: Thurs, Mar 5, 2020, 11:59 pm

This assignment must be submitted electronically via Gradescope andAutolab. Submit your homework as a `pdf` containing your written solutions. Submit your code as two .tar files: one containing each **.kyx** file and one containing each **.sig** and **.sml** file.

The following commands will create the correct tar files. The precise names of the tar files are not important, but the names of the files in each tar are. The autograder has some flexibility, but if it cannot find your problem files, it cannot grade them.

```
tar cvf handin_sml.tar inversion_calculus.sml inversion_calculus.sig

tar cvf handin_key.tar inotnotcontra2_Proof.kyx icovariance_Proof.kyx \
iScombinator_Proof.kyx iImplicortion_Proof.kyx \
iLazyApp_Proof.kyx iCPS_Proof.kyx iCNot_Proof.kyx
```

More succinctly (but potentially more error-prone if you have other things in the directory):

```
tar cvf handin_key.tar *_Proof.kyx
```

## 1 KeYmaera I

Prove the following statements in KeYmaera I:

**Task 1** (2 points)**. inotnotcontra2.kyx**

```
Definitions
  Bool p;
End.

Problem
  ((p | (p->false))->false)->false
End.
```

**Task 2** (2 points)**. icovariance.kyx**

```
Definitions
  Bool a;
  Bool b;
  Bool x;
  Bool y;
```

```
  Bool z;
End.

Problem
  (a -> b) -> (x -> y | a & z) -> (x -> y | b & z)
End.
```

**Task 3** (2 points)**. iScombinator.kyx**

```
Definitions
  Bool p;
  Bool q;
End.

Problem
  (p -> q) -> (p -> (q -> false)) -> (p -> false)
End.
```

**Task 4** (2 points)**. iImplicortion.kyx**

```
Definitions
  Bool p;
  Bool q;
  Bool r;
End.

Problem
  (p -> q) -> (p | r -> q | r)
End.
```

**Task 5** (2 points)**. iLazyApp.kyx**

```
Definitions
  Bool p;
End.

Problem
  p -> ((p -> false) -> false)
End.
```

**Task 6** (2 points)**. iCPS.kyx**

```
Definitions
  Bool a;
  Bool b;
End.

Problem
  (a -> b) -> a -> ((b -> false) -> false)
End.
```

**Task 7** (2 points)**. iCNot.kyx**

```
Definitions
  Bool a;
  Bool b;
  Bool c;
  Bool d;
End.

Problem
  ((((a -> false) -> false) -> false) ->
   (((b -> false) -> false) -> ((c -> false) -> d))  ->
          (a -> false) -> ((((b -> false) -> false) -> false) -> false) ->
         (((c -> false) -> false) -> false) -> ((d -> false) -> false)
  )
End.
```

## 2 Completeness Is Back

When we were working with natural deduction, we proved for each connective that given a derivation of that connective we can extend the derivation and reconstruct the connective. In sequent calculus we want to be able to show that, for any connective, it is legal to use an assumption of that connective as a verification of it even if that connective is not atomic. Put formally we want to show that the rule $\dfrac{}{\Gamma, A \Longrightarrow A}\ id$ is admissible.

**Task 8** (1 points)**.** Describe what inductive structure(s) we ought to induct on to show that the above rule is admissible.

**Task 9** (5 points)**.** Write the proof for the admissibility of this rule in the special case where $A$ is of the form $A_1 \vee A_2$.

## 3 Soundness too

In lecture and the lecture notes, we stated that we can prove the admissibility of cut via nested induction on the structure of the proof, and showed the cases for implication and conjunction, but left some cases unproven. Now, it is your turn to prove the admissibility of cut in a specific clause, or more precisely:

**Task 10** (6 points)**.** Extend the proof of the admissibility of cut from class by filling in the following inductive case.

**Case:** $\mathcal{D}$ ends in $\vee\mathsf{R}_2$ and $\mathcal{E}$ ends in $\vee\mathsf{L}$, where $\vee\mathsf{L}$ is applied on the principal formula of the cut.

## 4 Not Doing The Impossible

Aside from making proof search more reasonable, another great benefit of sequent calculus is that we can more easily prove that a proposition cannot be proven in a particular sequent calculus. In the following proof, you may appeal to the soundness of sequent calculus with respect to classical logic.

**Task 11** (5 points)**.** Show that $\Longrightarrow \neg(A \wedge B) \supset \neg A \vee \neg B$ cannot be proven in sequent calculus.

## 5 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 the rest of 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. 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.

$$\Delta \,;\, \Omega \xrightarrow{R} C \quad \text{Decompose } C \text{ on the right}$$
$$\Delta \,;\, \Omega \xrightarrow{L} C \quad \text{Decompose } \Omega \text{ on the left}$$

Above, $\Omega$ is an *ordered context* (say, a stack) that we only access at the right end. $\Delta$ 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**

$$\frac{}{\Delta, P\ ;\ \cdot \xrightarrow{L} P}\ \mathsf{init}\ (P \text{ atomic})$$

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

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

**Shift Rules**

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

**Search Rules**

$$\frac{\Delta\ ;\ \cdot \xrightarrow{R} A}{\Delta\ ;\ \cdot \xrightarrow{L} A \vee B}\ \vee R_1 \qquad \frac{\Delta\ ;\ \cdot \xrightarrow{R} B}{\Delta\ ;\ \cdot \xrightarrow{L} A \vee B}\ \vee R_2$$

## Proof search using inversion (no implication)

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 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 avoid this issue by implementing a proof search engine for the *implication-*

*free fragment* of the inversion calculus, where we omit the implication connective. In the next homework, you will extend this theorem prover to support implication using the **g4ip** calculus.

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

**Task 12.** 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] 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
```

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

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

- 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!