Informatics 1 – Introduction to Computation
Programming Project

due 12:00 Friday 28 November 2025

Good Scholarly Practice:

Please remember the good scholarly practice requirements of the University regarding work for credit. You can find guidance at the School page

http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct.

This also has links to the relevant University pages.

The Inf1A programming project is for learning, and — as an exception to the University's rules on good scholarly practice — you are encouraged to collaborate and to seek help from others when you get stuck. But you must always understand the solution you hand in, and be able to explain it to your tutor or a fellow student.

Please do not publish solutions to this exercise on the internet or elsewhere, to avoid others copying your solution.

1 Sequent calculus

Your task is to implement the sequent calculus in Haskell. In the mandatory part of the project, you will omit implication $\to$ and bi-implication $\leftrightarrow$ and construct proofs as explained on pages 126–127 of the textbook using the rules given there:

$$\frac{}{\Gamma, a \vDash a, \Delta}\ I$$

$$\frac{\Gamma \vDash a, \Delta}{\Gamma, \neg a \vDash \Delta}\ \neg L \qquad \frac{\Gamma, a \vDash \Delta}{\Gamma \vDash \neg a, \Delta}\ \neg R$$

$$\frac{\Gamma, a, b \vDash \Delta}{\Gamma, a \wedge b \vDash \Delta}\ \wedge L \qquad \frac{\Gamma \vDash a, \Delta \qquad \Gamma \vDash b, \Delta}{\Gamma \vDash a \wedge b, \Delta}\ \wedge R$$

$$\frac{\Gamma, a \vDash \Delta \qquad \Gamma, b \vDash \Delta}{\Gamma, a \vee b \vDash \Delta}\ \vee L \qquad \frac{\Gamma \vDash a, b, \Delta}{\Gamma \vDash a \vee b, \Delta}\ \vee R$$

In the optional part, you will add rules for $\to$ and $\leftrightarrow$.

Recall that proofs start with a sequent and are constructed by applying rules, working upwards, until you reach **simple premises** involving only "bare" predicates, not containing any connectives. The proof shows that the sequent you started with follows from those premises. If the proof ends in the empty set of simple premises, you have shown that the sequent you started with is **universally valid**.

Here's an example of a sequent calculus proof, which proves that $\vDash ((\neg p \vee q) \wedge \neg p) \vee p$ is universally valid:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \vDash q, p}\ I}{\vDash \neg p, q, p}\ \neg R}{\vDash \neg p \vee q, p}\ \vee R \qquad \dfrac{\dfrac{}{p \vDash p}\ I}{\vDash \neg p, p}\ \neg R}{\vDash (\neg p \vee q) \wedge \neg p, p}\ \wedge R}{\vDash ((\neg p \vee q) \wedge \neg p) \vee p}\ \vee R$$

Here's another example, which proves that $\vDash \neg((\neg a \vee b) \wedge (\neg c \vee b)) \vee (\neg a \vee c)$ follows from $a, b \vDash c$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg a, \neg c \vee b \vDash \neg a, c}\ I \qquad \dfrac{\dfrac{\dfrac{\dfrac{a, b \vDash c}{b \vDash \neg a, c}\ \neg R}{b, \neg c \vDash \neg a, c}\ \neg L \qquad \dfrac{a, b \vDash c}{b \vDash \neg a, c}\ \neg R}{b, \neg c \vee b \vDash \neg a, c}\ \vee L}{\neg a \vee b, \neg c \vee b \vDash \neg a, c}\ \vee L}{(\neg a \vee b) \wedge (\neg c \vee b) \vDash \neg a, c}\ \wedge L}{(\neg a \vee b) \wedge (\neg c \vee b) \vDash \neg a \vee c}\ \vee R}{\vDash \neg((\neg a \vee b) \wedge (\neg c \vee b)), \neg a \vee c}\ \neg R}{\vDash \neg((\neg a \vee b) \wedge (\neg c \vee b)) \vee (\neg a \vee c)}\ \vee R$$

The number of times that a premise appears is immaterial, because we're interested in the *set* of premises from which the conclusion holds. In this proof, the premise $a, b \vDash c$ happens to appear twice.

There's often more than one proof starting from a given sequent, but all end with the same set of premises. Because each rule of the sequent calculus eliminates one connective from an antecedent or a succedent, proofs always terminate.

2 Implementing sequent calculus in Haskell

To implement the sequent calculus in Haskell, you will start with an algebraic datatype similar to the one in FP Tutorial 6 that defines propositions with variable names of type `String`:

```
type Name = String

data Prop = Var Name
          | Not Prop
          | Prop :||: Prop
          | Prop :&&: Prop
          | Prop :->: Prop
          | Prop :<->: Prop
```

The last two cases will be required only for the optional part but they are included from the start to allow the optional part to be coded as a simple extension of the mandatory part.

Sequents are defined as follows:

```
data Sequent = [Prop] :|=: [Prop]

infix 0 :|=:
```

We use unordered lists of propositions, possibly with repetitions, to represent the sets of antecedents and succedents of sequents.

You are provided with a file `Sequent.hs` containing the above type definitions together with code for declaring `Prop` and `Sequent` as instances of `Show` and (for purposes of QuickCheck test case generation) of `Arbitrary`. If you want to use QuickCheck to test your code (strongly recommended! always test your code!) then you will need to comment out the following two lines for generating test cases that include $\rightarrow$ and $\leftrightarrow$ until you get to the optional part:

```
, liftM2 (:->:) p2 p2
, liftM2 (:<->:) p2 p2
```

Don't make any other changes to `Sequent.hs`.

Also provided is a skeleton file called `Project.hs` which imports `Sequent`. Your job is to define the function

```
prove :: Sequent -> [Sequent]
```

in `Project.hs` which, when applied to a sequent, produces the list of simple sequents from which that sequent follows according to a sequent calculus proof. You need not produce the proof itself, just the list of premises.

For the first example above, it should produce

```
> p = Var "p"
> q = Var "q"
> prove ([] :|=: [((Not p :||: q) :&&: Not p) :||: p])
[]
```

For the second example, one correct result would be

```
> a = Var "a"
> b = Var "b"
> c = Var "c"
> prove ([] :|=: [(Not ((Not a :||: b) :&&: (Not c :||: b))) :||: (Not a :||: c)])
[[a,b] :|=: [c]]
```

Because of our use of unordered lists to represent sets, another correct result would be

```
> prove ([] :|=: [(Not ((Not a :||: b) :&&: (Not c :||: b))) :||: (Not a :||: c)])
[[a,b] :|=: [c,c],[b,a,b] :|=: [c]]
```

and there are many others.

3 How to proceed

How you proceed is up to you. That's why this is a project and not a tutorial exercise.

One way to go is to start by defining the rules of sequent calculus as Haskell functions, with for instance

```
orL :: Sequent -> Maybe [Sequent]
```

Note that the result type is `Maybe [Sequent]`. We use `[Sequent]` rather than `Sequent` because the $\vee L$ rule has more than one premise. And we use `Maybe [Sequent]` rather than `[Sequent]` because $\vee L$ isn't applicable to all sequents, only ones having an antecedent with $\vee$ as its main connective.

For the $\vee R$ rule, you could define a function

```
orR :: Sequent -> Maybe Sequent
```

since it can produce at most one premise, but it's probably better to use the same type for all rules for the sake of the next step:

```
orR :: Sequent -> Maybe [Sequent]
```

You might even want to define a type for rules:

```
type Rule = Sequent -> Maybe [Sequent]
orL, orR :: Rule
```

Hint: The main work in applying a rule to a sequent involves finding appropriate items in the antecedent list and/or the succedent list and replacing them with new items. You might find useful functions for doing this in `Data.List`.

Now you can use the definitions of the rules to search for a proof. Given a sequent, `prove` tries rules until one applies (that is, it produces `Just` *seqs* instead of `Nothing`). Then do that to each of the sequents in *seqs*. Keep going until none of the rules applies. The result is a list of simple sequents from which the original sequent follows.

The definition of `prove` will probably rely on helper functions that try individual rules and/or lists of rules. How you do this is up to you.

4 Optional Material: Adding implication and bi-implication

> Following the Common Marking Scheme, a student with good mastery of the material is expected to get 3/4 points. This section is for demonstrating exceptional mastery of the material. It is optional and worth 1/4 points.

The definitions of the types `Prop` and `Sequent` already allow for them to contain implication $\rightarrow$ and bi-implication $\leftrightarrow$.

Here are the sequent calculus rules for these connectives from page 222 of the textbook:

$$\frac{\Gamma \vDash a, \Delta \qquad \Gamma, b \vDash \Delta}{\Gamma, a \rightarrow b \vDash \Delta} \rightarrow L \qquad \frac{\Gamma, a \vDash b, \Delta}{\Gamma \vDash a \rightarrow b, \Delta} \rightarrow R$$

$$\frac{\Gamma, a \rightarrow b, b \rightarrow a \vDash \Delta}{\Gamma, a \leftrightarrow b \vDash \Delta} \leftrightarrow L \qquad \frac{\Gamma \vDash a \rightarrow b, \Delta \qquad \Gamma \vDash b \rightarrow a, \Delta}{\Gamma \vDash a \leftrightarrow b, \Delta} \leftrightarrow R$$

Here is a proof that $a \rightarrow b \vDash \neg b \rightarrow \neg a$ is universally valid, using these rules and the ones given earlier:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg b, a \vDash a}\, I}{\neg b \vDash a, \neg a} \neg R \qquad \dfrac{\dfrac{}{b \vDash b, \neg a}\, I}{b, \neg b \vDash \neg a} \neg L}{a \rightarrow b, \neg b \vDash \neg a} \rightarrow L}{a \rightarrow b \vDash \neg b \rightarrow \neg a} \rightarrow R$$

For the optional part, extend your function

```
prove :: Sequent -> [Sequent]
```

to do sequent calculus proofs of sequents involving propositions that may contain implication and/or bi-implication. Applying `prove` to a sequent that doesn't contain implication or bi-implication should produce the same result as before. If you do the optional part, you don't have to submit two versions of `prove`; the extended version, that works for implications and bi-implications as well as the other connectives, will suffice.

For the example above, it should produce

```
> a = Var "a"
> b = Var "b"
> prove (a :->: b :|=: Not b :->: Not a)
[]
```

If you use QuickCheck for testing your code, remember to restore the two lines of the declaration in `Sequent.hs` that `Prop` is an instance of `Arbitrary` so that it will also generate test cases including `:->:` and `:<->:`.

5 Challenge: Shorter proofs

> Challenges are meant to be difficult. You can receive full marks without attempting the challenge.

From a given sequent, there are often different proofs leading to the same set of premises. Your challenge here is to find the shortest one.

Instrument your `prove` function to count how many rule applications the proof involved, giving a function

```
proveCount :: Sequent -> ([Sequent], Int)
```

such that

```
prove seq == fst (proveCount seq)
```

(where equality takes account of the fact that sets are represented as unordered lists, possibly with repetitions) and `snd (proveCount seq)` is the number of rule applications involved in that proof.

For example, the first proof in Section 1 involves 7 rule applications, the second proof there involves 10 rule applications, and the proof in Section 4 involves 6 rule applications.

Make changes to the way you do proof search, if necessary, to make the number of rule applications as small as possible.

(Is there a way to change proof search to make proofs as *long* as possible?)

6 Marking

The programming project will be marked by your tutor, and is worth 20% of the mark for Inf1A.

Your mark on the scale 0–4 will be based mainly on the following tests. (Your tutor will also check your code to make sure, among other things, that it is not written to pass *only* these tests.) The tests have been set up in the automarker. They will be run every time you submit `Project.hs`, and you will be able to check the results. You can submit as often as you like, but only the last submission before the deadline will be taken into account in the marking.

The sequents in the tests below have been typeset to make them a little easier to read, but of course the actual tests use Haskell syntax. Because of our use of unordered lists to represent tests, lists of assumptions that are equivalent to those shown as results are also correct.

To get 1 point, it's enough that any one of the following one-step proofs is correct. Non-working code that shows some sensible work may also be awarded 1 point.

```
> prove ( ⊨ a ∨ b )    -- ∨R
[ ⊨ a, b ]
> prove ( a ∨ b ⊨ )    -- ∨L
[ a ⊨ , b ⊨ ]
> prove ( ⊨ a ∧ b )    -- ∧R
 [ ⊨ a , ⊨ b ]
> prove ( a ∧ b ⊨ )    -- ∧L
[ a, b ⊨ ]
> prove ( ⊨ ¬a )       -- ¬R
[ a ⊨ ]
> prove ( ¬a ⊨ )       -- ¬L
[ ⊨ a ]
> prove ( a, b ⊨ b, c ) -- I
[ ]
```

To get 2 points, the following proof, which uses only the non-branching rules, must also be correct.

```
> prove ( ⊨ (¬(e ∧ (f ∧ ¬¬c))) ∨ (¬a ∨ c) )
[ ]
```

To get 3 points, most or all of the following proofs must also be correct.

```
> prove ( ⊨ ((¬a ∨ b) ∧ ¬a) ∨ a )
[ ]
> prove ( ⊨ ¬((¬a ∨ b) ∧ (¬c ∨ b)) ∨ ¬a ∨ c )
[ a, b ⊨ c ]
> prove ( ¬c ∨ (f ∨ b), (a ∨ d) ∧ (b ∨ b) ⊨ ¬(d ∨ b), ¬c ∨ (f ∧ e), (f ∧ b) ∨ (c ∧ e), ¬(a ∨ c), ¬e ∧ (c ∨ e), ¬¬e )
[ ]
> prove ( (b ∨ f) ∨ (d ∨ c), ¬(d ∨ a), ¬(d ∧ f), ¬(a ∧ f), (a ∨ e) ∧ (b ∨ c), ¬¬c ⊨ )
[ b, c, e ⊨ a, d , b, c, e ⊨ a, d, f , b, c, e, f ⊨ a, d , c, e ⊨ a, d , c, e ⊨ a, d, f , c, e, f ⊨ a, d ]
> prove ( ¬¬b, (d ∨ b) ∨ ¬c, ¬(d ∧ f)
                  ⊨ (b ∨ d) ∧ (c ∧ e), (a ∨ c) ∨ ¬f, ¬(f ∧ f), (e ∧ d) ∧ (a ∨ d), ¬¬c, (b ∧ b) ∧ (e ∨ b) )
[ ]
> prove ( (f ∨ a) ∧ ¬f, (a ∨ a) ∧ (c ∨ b), ¬a ∨ ¬c, ¬d ∨ (a ∨ d)
                                        ⊨ (f ∧ d) ∨ (d ∨ d), (c ∨ e) ∨ (d ∨ e), (b ∨ a) ∧ (f ∨ e) )
[ a, b ⊨ c, d, e, f ]
> prove ( (a ∨ b) ∨ ¬d, (e ∨ b) ∧ ¬f ⊨ ¬¬a, (f ∧ f) ∨ (f ∧ f), ¬f ∧ (e ∨ c), (f ∧ b) ∧ (e ∨ a) )
[ b ⊨ a, e, c, f , b ⊨ a, c, d, e, f ]
> prove ( (a ∨ f) ∨ (c ∨ d), (b ∧ f) ∨ (f ∨ a), (f ∨ f) ∧ ¬f, ¬a ∨ (c ∨ e) ⊨ ¬e ∧ ¬c, (b ∧ c) ∨ ¬a, ¬(e ∨ d) )
[ ]
> prove ( ⊨ (d ∨ b) ∨ (e ∧ a), ¬f ∧ (d ∧ b), (c ∧ b) ∧ ¬e, (c ∨ c) ∨ (d ∨ d),
                                        ¬e ∧ (a ∧ e), (a ∨ e) ∨ (c ∨ d), (d ∨ d) ∧ ¬e, (c ∧ d) ∧ ¬a, ¬(e ∨ c) )
[ ]
```

To get 4 points, most or all of the following proofs involving implication and/or bi-implication must also be correct.

```
> prove ( a → b ⊨ ¬b → ¬a )
[ ]
> prove ( (c ∨ d) → (d ∧ f), (a ↔ a) ↔ (b ∨ a) ⊨ (c ↔ b) ↔ (b → a), ¬(b ∨ d), ¬(b ∧ d), (d ↔ f) → (c ∨ e) )
[ b, a, f, d ⊨ c, e ]
> prove ( ¬(f ↔ f), ¬f ↔ (d ↔ e), (d → f) ↔ (a ∨ b) ⊨ (d ∧ c) → (c ↔ d), ¬(c ↔ e) )
[ ]
> prove ( (e → b) → (a → f), (b ∨ a) ∨ ¬f, (d ↔ b) ↔ (f ∧ e) ⊨ (d ↔ d) ∨ (e ↔ c), ¬(d → f) )
[ ]
> prove ( (f ∨ b) ∧ (b → d), (e ∨ d) ∨ ¬e, (f → e) → ¬f, (e ∨ d) ∧ (d ↔ c), ¬(c ↔ b), ¬f ∧ (a ∧ a),
                                        (f ↔ c) → (c ∨ f) ⊨ (e ∧ f) ∨ (f ∧ a), ¬f ∧ ¬d, (f ∨ c) ↔ (f ↔ b) )
[ ]
> prove ( (e ∧ f) ↔ (a ∧ e), (b ∨ d) ∧ (f ↔ c), ¬(d ↔ f), (d ∧ e) ∨ (c → f), (e ↔ f) ↔ (b ∨ d) ⊨ (e ∨ d) ∧ ¬e )
[ f, e, a, b, c ⊨ d ]
```

The challenge part isn't worth any points, but the following tests are included in the automarker. Correct for these means getting the same assumptions with a proof that is no longer than the number of rules shown, which is achieved by the sample solution.

```
> proveCount ( (a ∨ f) ∨ (c ∨ d), (b ∧ f) ∨ (f ∨ a), (f ∨ f) ∧ ¬f, ¬a ∨ (c ∨ e) ⊨ ¬e ∧ ¬c, (b ∧ c) ∨ ¬a, ¬(e ∨ d) )
([ ], 168)
> proveCount ( ¬c ∨ (f ∨ b), (a ∨ d) ∧ (b ∨ b) ⊨ ¬(d ∨ b), ¬c ∨ (f ∧ e), (f ∧ b) ∨ (c ∧ e), ¬(a ∨ c), ¬e ∧ (c ∨ e), ¬¬e )
([ ], 39)
> proveCount ( ¬¬b, (d ∨ b) ∨ ¬c, ¬(d ∧ f)
                ⊨ (b ∨ d) ∧ (c ∧ e), (a ∨ c) ∨ ¬f, ¬(f ∧ f), (e ∧ d) ∧ (a ∨ d), ¬¬c, (b ∧ b) ∧ (e ∨ b) )
([ ], 58)
> proveCount ( (c ∨ d) → (d ∧ f), (a ↔ a) ↔ (b ∨ a) ⊨ (c ↔ b) ↔ (b → a), ¬(b ∨ d), ¬(b ∧ d), (d ↔ f) → (c ∨ e) )
([ a, b, f, d ⊨ c, e ], 144)
> proveCount ( ¬(f ↔ f), ¬f ↔ (d ↔ e), (d → f) ↔ (a ∨ b) ⊨ (d ∧ c) → (c ↔ d), ¬(c ↔ e) )
([ ], 105)
> proveCount ( (e → b) → (a → f), (b ∨ a) ∨ ¬f, (d ↔ b) ↔ (f ∧ e) ⊨ (d ↔ d) ∨ (e ↔ c), ¬(d → f) )
([ ], 306)
> proveCount ( (f ∨ b) ∧ (b → d), (e ∨ d) ∨ ¬e, (f → e) → ¬f, (e ∨ d) ∧ (d ↔ c), ¬(c ↔ b), ¬f ∧ (a ∧ a),
                                        (f ↔ c) → (c ∨ f) ⊨ (e ∧ f) ∨ (f ∧ a), ¬f ∧ ¬d, (f ∨ c) ↔ (f ↔ b) )
([ ], 229)
> proveCount ( (e ∧ f) ↔ (a ∧ e), (b ∨ d) ∧ (f ↔ c), ¬(d ↔ f), (d ∧ e) ∨ (c → f), (e ↔ f) ↔ (b ∨ d) ⊨ (e ∨ d) ∧ ¬e )
([ a, b, c, e, f ⊨ d ], 491)
```