MATH230: Tutorial XY

Proofs-as-Programs: Minimal Logic in L∃∀N

Key ideas
- Write minimal logic proofs as programs.
- Write explicit proof-terms.
- Write proofs using tactic mode.

Relevant lectures: Lectures x, y, and z
Relevant reading: Theorem Proving in L∃∀N 4 : Propositions and Proofs

Hand in exercises:
Due following Friday @ 5pm to the tutor, or lecturer.

Discussion Questions

1. Show $P \vdash \neg\neg P$.

2. Show $P \to Q \vdash P \to (P \wedge Q)$.

3. Show $(P \wedge Q) \vee R \ \vdash \ (P \vee R) \wedge (Q \vee R)$.

Tutorial Exercises

1. **Minimal Logic in L∃∀N**

   (a) $(P \wedge Q) \to R \dashv\vdash P \to (Q \to R)$

   This proof has been written in both directions separately. These are combined into a proof of the iff at the end. Notice the comments, indicated by lines with -- at the beginning, show how the sequent syntax that we have been using throughout the course lines up with the syntax of L∃∀N.

```
--                  (P ∧ Q) → R  ⊢ P → (Q → R)
theorem curry (f : (P ∧ Q) → R) : P → (Q → R) :=
  λ wp : P =>
   λ wq : Q =>
     f (And.intro wp wq)

--                    P → (Q → R)  ⊢ (P ∧ Q) → R
theorem uncurry (f : P → (Q → R)) : (P ∧ Q) → R :=
  λ wpq : P ∧ Q =>
   (f wpq.left) (wpq.right)

--                  P → (Q → R) ⊣⊢ (P ∧ Q) → R
theorem currying : P → (Q → R) ↔ (P ∧ Q) → R :=
  Iff.intro
   (λ f : P → (Q → R) =>
      λ wpq : P ∧ Q =>
        (f wpq.left) (wpq.right))
   (λ f : (P ∧ Q) → R =>
     λ wp : P =>
      λ wq : Q =>
        f (And.intro wp wq))
```

   Proofs are programs and therefore can be factored like any other program! Since the proofs we write are programs, they can be called upon in later proofs. This means that the proof of the iff can be simplified by calling the proofs of each direction.

```
--                 P → (Q → R) ⊣⊢ (P ∧ Q) → R
theorem crrying : P → (Q → R)  ↔ (P ∧ Q) → R :=
Iff.intro
   (uncurry P Q R)
   (curry P Q R)
```

With L∃∀N one has the option of using meta-programs (so called *tactics*) to help write proof terms, rather than explicitly writing the proof terms oneself. This is useful now, often shortening proofs, but becomes indispensable later when writing proofs of even slightly non-trivial mathematics. It requires some getting used to, often this means working backwards - so it's worth practicing on these proofs.

Here are the proofs about currying, but written using tactics instead of providing the proof term explicitly. Tactic proofs still produce proof terms - that is how a proof is verified. One can use the "#print" command to see the proof term obtained from the tactic proof. It will probably be similar, if not exactly the same, to your original proof term.

```
--                  (P ∧ Q) → R  ⊢ P → (Q → R)
theorem curry (f : (P ∧ Q) → R) : P → (Q → R) :=
by
   intro p q
   exact f (And.intro p q)

--                    P → (Q → R)  ⊢ (P ∧ Q) → R
theorem uncurry (f : P → (Q → R)) : (P ∧ Q) → R :=
by
   intro h
   exact (f h.left) h.right

--                  P → (Q → R) ⊣⊢ (P ∧ Q) → R
theorem currying : P → (Q → R)  ↔ (P ∧ Q) → R :=
by
   apply Iff.intro
   apply uncurry
   apply curry
```

(b) $\neg P \vee \neg Q \vdash \neg(P \wedge Q)$

```
--                                        ¬P ∨ ¬Q  ⊢ ¬(P ∧ Q)
theorem deMogran_notConj_factor (t : ¬P ∨ ¬Q) : ¬(P ∧ Q) :=
  λ w : P ∧ Q =>
   Or.elim t
     (λ np : ¬P =>
      np w.left)
     (λ nq : ¬Q =>
      nq w.right)
```

```
--                                        ¬P ∨ ¬Q  ⊢ ¬(P ∧ Q)
theorem deMogran_notConj_factor (t : ¬P ∨ ¬Q) : ¬(P ∧ Q) :=
by
   intro h
   apply Or.elim t
   intro np
   exact np h.left
   intro nq
   exact nq h.right
```

(c) $\neg(P \vee Q) \dashv\vdash \neg P \wedge \neg Q$

```
--                        ¬(P ∨ Q) ⊢ ¬P ∧ ¬Q
--                                   ⊢ ¬(P ∨ Q) → ¬P ∧ ¬Q
theorem deMorgan_notDisj_expand : ¬(P ∨ Q) → ¬P ∧ ¬Q :=
  λ f : ¬(P ∨ Q) =>
   And.intro (λ wp : P =>
            f (Or.intro_left Q wp))
           (λ wq : Q =>
            f (Or.intro_right P wq))

--                        ¬P ∧ ¬Q ⊢ ¬(P ∨ Q)
--                                   ⊢ ¬P ∧ ¬Q → ¬(P ∨ Q)
theorem deMorgan_notDisj_factor : ¬P ∧ ¬Q → ¬(P ∨ Q) :=
  λ t1 : ¬P ∧ ¬Q =>
   λ t2 : P ∨ Q =>
     Or.elim t2
      (λ wp : P =>
        t1.left wp)
      (λ wq : Q =>
        t1.right wq)

--                          ⊢ ¬(P ∨ Q) ↔ ¬P ∧ ¬Q
theorem deMorgan_notDisj : ¬(P ∨ Q) ↔ ¬P ∧ ¬Q :=
  Iff.intro
   (deMorgan_notDisj_expand P Q)
   (deMorgan_notDisj_factor P Q)


--                        ¬(P ∨ Q) ⊢ ¬P ∧ ¬Q
--                                   ⊢ ¬(P ∨ Q) → ¬P ∧ ¬Q
theorem deMorgan_notDisj_expand : ¬(P ∨ Q) → ¬P ∧ ¬Q :=
by
   intro h
   apply And.intro
   intro p
   have t1 := Or.intro_left Q p
   exact h t1
   intro q
   have t2 := Or.intro_right P q
   exact h t2

--                        ¬P ∧ ¬Q ⊢ ¬(P ∨ Q)
--                                   ⊢ ¬P ∧ ¬Q → ¬(P ∨ Q)
theorem deMorgan_notDisj_factor : ¬P ∧ ¬Q → ¬(P ∨ Q) :=
by
   intro t1 t2
   apply Or.elim t2
   intro p
   exact t1.left p
   intro q
   exact t1.right q

--                          ⊢ ¬(P ∨ Q) ↔ ¬P ∧ ¬Q
theorem deMorgan_notDisj : ¬(P ∨ Q) ↔ ¬P ∧ ¬Q :=
by
   apply Iff.intro
   exact deMorgan_notDisj_expand P Q
   exact deMorgan_notDisj_factor P Q
```

(d) $P \to Q,\ Q \to R \vdash P \to R$

```
--                                P → Q,      Q → R  ⊢ P → R
theorem impl_composition (f : P → Q) (g : Q → R) : P → R :=
  λ wp : P =>
   g (f wp)
```

```
--                                P → Q,      Q → R  ⊢ P → R
theorem impl_composition (f : P → Q) (g : Q → R) : P → R :=
by
   intro p
   exact g (f p)
```

(e) $P \lor Q, \quad P \to R, \quad Q \to S \vdash R \lor S$

```
theorem case (t : P ∨ Q) (f : P → R) (g : Q → S) : R ∨ S :=
Or.elim t
  (λ wp : P =>
   Or.intro_left S (f wp))
  (λ wq : Q =>
   Or.intro_right R (g wq))


--                P ∨ Q,       P → R,       Q → S  ⊢ R ∨ S
theorem case (t : P ∨ Q) (f : P → R) (g : Q → S) : R ∨ S :=
  by
   apply Or.elim t
   intro p
   exact Or.intro_left S (f p)
   intro q
   exact Or.intro_right R (g q)
```

(f) $\neg R \vee \neg D, \quad P \to R, \quad Q \to D \vdash \neg P \vee \neg Q$

```
--            ¬R ∨ ¬S,       P → R,       Q → S  ⊢ ¬P ∨ ¬Q
example (t1 : ¬R ∨ ¬S) (f : P → R) (g : Q → S) : ¬P ∨ ¬Q :=
  Or.elim t1
   (λ nr : ¬R =>
     Or.intro_left (¬Q)
      (λ wp : P =>
        nr (f wp)))
   (λ ns : ¬S =>
     Or.intro_right (¬P)
      (λ wq : Q =>
        ns (g wq)))
```

Alternatively, we can factor out the proof of modus tollens.

```
--                          P → Q,        ¬Q  ⊢ ¬P
theorem modus_tollens (f : P → Q) (nq : ¬Q) : ¬P :=
  λ wp : P =>
   nq (f wp)
```

```
--            ¬R ∨ ¬S,       P → R,       Q → S  ⊢ ¬P ∨ ¬Q
example (t1 : ¬R ∨ ¬S) (f : P → R) (g : Q → S) : ¬P ∨ ¬Q :=
  Or.elim t1
   (λ nr : ¬R =>
     Or.intro_left (¬Q)
                (modus_tollens P R f nr))
   (λ ns : ¬S =>
     Or.intro_right (¬P)
                (modus_tollens Q S g ns))
```

```
--            ¬R ∨ ¬S,       P → R,       Q → S  ⊢ ¬P ∨ ¬Q
example (t1 : ¬R ∨ ¬S) (f : P → R) (g : Q → S) : ¬P ∨ ¬Q :=
by
   cases t1 with
   | inr h2 => have nq : ¬Q := modus_tollens Q S g h2
               apply Or.intro_right (¬P) nq
   | inl h1 => have np : ¬P := modus_tollens P R f h1
               apply Or.intro_left (¬Q) np
```

(g) $P,\ \ \neg P \vdash \neg Q$

```
--                              P,        ¬P  ⊢ ¬Q
theorem minimal_falso (p : P) (np : ¬P) : ¬Q :=
  λ wq : Q =>
   np p
```

```
--                              P,        ¬P  ⊢ ¬B
theorem minimal_falso (p : P) (np : ¬P) : ¬B :=
by
   intro b
   exact np p
```

(h) $P \to Q,\ P \to \neg Q \vdash \neg P$

```
--                          P → Q       P → ¬Q  ⊢ ¬P
theorem lesser_falso (f : P → Q) (g : P → ¬Q) : ¬P :=
λ wp : P =>
   (g wp) (f wp)
```

```
--                          P → Q       P → ¬Q  ⊢ ¬P
theorem lesser_falso (f : P → Q) (g : P → ¬Q) : ¬P :=
by
   intro p
   exact (g p) (f p)
```