MATH230: Tutorial Nine (Solutions)

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

Key ideas
- Learn L∃∀N syntax.
- Write proof terms in L∃∀N.
- Compare syntax across languages.

Relevant lectures:
Relevant reading: Theorem Proving in L∃∀N 4

Hand in exercises: 1
Sue Friday @ 5pm to the tutor, or to lecturer.

Discussion Questions

1. In a previous tutorial we wrote a proof-term witnessing the sequent

$$P \to Q \ \vdash \ P \to (P \wedge Q)$$

   Translate this proof-term into the syntax for L∃∀N 4.

2. Write a proof-term in L∃∀N 4 to prove the following sequent

$$P \vdash \neg\neg P$$

3. Write a proof-term in L∃∀N 4 to prove the following sequent

$$(P \wedge Q) \vee R \ \vdash \ (P \vee R) \wedge (Q \vee R)$$

Tutorial Exercises

1. Throughout the course we have introduced rules of inference in logic, type constructors and destructors in the typed λ-calculus, and now we are to see how they are implemented in L$\exists\forall$N. As you write proof-terms for the sequents in Exercise 2 enter the missing components in the following table:

PL	λ	L$\exists\forall$N 4
$\wedge I$		
$\wedge E_r$		
$\wedge E_l$		
$\rightarrow I$	$\lambda \_ : \_ .$	$\lambda \_ : \_ =>$
$\rightarrow E$		
$\vee I_r$		
$\vee I_l$		
$\vee E$		

Table 1: Syntax of logic, λ-calculus, and L$\exists\forall$N 4

2. For each of the sequents below, write proof-terms in L$\exists\forall$N 4. You have seen these proofs before, so it is a translating into L$\exists\forall$N exercise, rather than proving anything new.

   (a) $P \rightarrow Q, \neg Q \vdash \neg P$

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

   (b) $(P \wedge Q) \rightarrow R \dashv\vdash P \rightarrow (Q \rightarrow R)$

   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$\exists\forall$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 currying : P → (Q → R)  ↔ (P ∧ Q) → R :=
Iff.intro
   (uncurry P Q R)
   (curry P Q R)
```

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

```
--                                          ¬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)
```

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

```
--                          ¬(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)
```

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

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

(f) $P \vee Q,\ \ P \to R,\ \ Q \to S \vdash R \vee S$

```
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))
```

(g) $P \to R,\ \ Q \to S,\ \ \neg R \vee \neg S \vdash \neg P \vee \neg Q$

```
--            ¬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)))
```

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

```
--            ¬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 from above. Each of these theorems you are proving are programs. This means they can be called in other proofs - just as we have been recycling proofs in our natural deductions. Here is the same proof, but with Modus Tollens factored out.

```
--            ¬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))
```

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

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

3. For each of the sequents below write proof-terms in L∃∀N. These exercises have not appeared in earlier tutorials. You could prove these by-hand first and then translate that into L∃∀N. Or you could write the proof straight into L∃∀N.

(a) $P \vdash \neg\neg P$

```
--                                P  ⊢ ¬¬P
theorem double_neg_intro (wp : P) : ¬¬P :=
λ wnp : ¬P =>
   wnp wp
```

(b) $\neg\neg\neg P \vdash \neg P$

```
--                           ¬¬¬P  ⊢ ¬P
theorem minmial_dne (f : ¬¬¬P) : ¬P :=
λ wp : P =>
   f (double_neg_intro P wp)
```

(c) $\vdash \neg\neg(P \vee \neg P)$

```
--                          ⊢ ¬¬(P ∨ ¬P)
theorem cant_refute_lem : ¬¬(P ∨ ¬P) :=
   λ g : ¬(P ∨ ¬P) =>
   g (Or.intro_right (P)
```

```
(λ x : P =>
   g (Or.intro_left (¬P) x)))
```