CIS 194: Homework 8

Due Wednesday, 8 April

Propositional Logic

In this section, you will prove some theorems in *Propositional Logic*, using the Haskell compiler to verify your proofs. The Curry-Howard isomorphism states that types are equivalent to theorems and programs are equivalent to proofs. There is no need to run your code or even write test cases; once your code type checks, the proof will be accepted!

Implication

We will begin by defining some *logical connectives*. The first logical connective is implication. Luckily, implication is already built in to the Haskell type system in the form of arrow types. The type `p -> q` is equivalent to the logical statement $P \to Q$. We can now prove a very simple theorem called Modus Ponens. Modus Ponens states that if P implies Q and P is true, then Q is true. It is often written as:

$$\frac{P \to Q \qquad P}{Q}$$

Figure 1: Modus Ponens

We can write this down in Haskell as well:

```
modus_ponens :: (p -> q) -> p -> q
modus_ponens pq p = pq p
```

Notice that the type of `modus_ponens` is identical to the Prelude function `($)`. We can therefore simplify the proof[1]:

[1] This is a very interesting observation. The logical theorem Modus Ponens is exactly equivalent to function application!

```
modus_ponens :: (p -> q) -> p -> q
modus_ponens = ($)
```

Disjunction

The disjunction (also known as "or") of the propositions P and Q is often written as $P \vee Q$. In Haskell, we will define disjunction as follows:

```
-- Logical Disjunction
data p \/ q = Left  p
            | Right q
```

The disjunction has two constructors, `Left` and `Right`. The `Left` constructor takes in something of type `p` and the `Right` constuctor takes in something of type `q`. This means that a proposition of type `p \/ q` can be either a `p` or a `q` where `p` and `q` are themselves propositions. This type is actually identical to Haskell's `Either`, but we redefine it here with a nicer syntax.

Conjunction

Now let's take a look at conjunctions ("and"). In logic, the conjunction of P and Q is written as $P \wedge Q$. In Haskell it is written as:

```
-- Logical Conjunction
data p /\ q = Conj p q
```

Figure 2: Schoolhouse Rock made an episode about conjunctions which featured the phrase "Conjunction junction, what's your function?" Little did they know that conjunctions are implemented as a data type, not a function.

Unlike disjunction, conjunction only has one constructor. This is because there is only one way to have a conjunction proposition; both `p` and `q` must be true. This type is equivalent to Haskell's tuple.

Bottom and Logical Negation

In order for a logic to be consistent, there must be some proposition that is not provable. This is usually written as $\bot$ (pronounced bottom). In our Haskell formulation of propositional logic, we will call it `False` and define it as follows.

```
data False
```

`False` is a datatype that has no constructors. This means that the type `False` in *uninhabited*; it is not provable since there is no way to write a program of type `False`. Now that we have a notion of `False`, we can define the logical connective "not" (written as $\neg P$):

```
type Not p = p -> False
```

The `Not` type is really just an alias for `p -> False`. In other words, if `p` were true, then `False` would also be true. Now, let's prove another theorem, Modus Tollens. Modus Tollens states that if P imples Q and Q is not true, then P is not true. In logic, it is written as:

$$\frac{P \to Q \qquad \neg Q}{\neg P}$$

Figure 3: Modus Tollens

Now, let's look at the Haskell version of Modus Tollens:

```
modus_tollens :: (p -> q) -> Not q -> Not p
modus_tollens pq not_q = not_q . pq
```

This is a bit more involved than the proof of Modus Ponens above. Recall that `Not p` is really just a handy syntax for `p -> False`. This means that the proof of Modus Tollens should be a function that takes in an inhabitant of the proposition `p` and derives a contradiction from it. This is not too hard to do since we assume that `p` implies `q` and `q` implies `False`.

If and Only If

We define one last logical connective: if and only if. If and only if is also known as as the biconditional connective since it is equivalent to $P \to Q$ and $Q \to P$. If and only if is usually written as $P \leftrightarrow Q$. In Haskell we use the following syntax:

```
type p <-> q = (p -> q) /\ (q -> p)
```

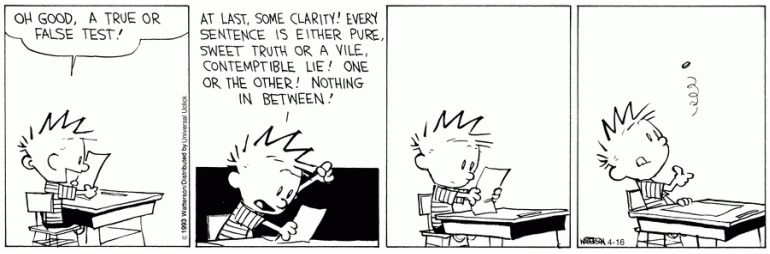

Figure 4: The Law of Excluded Middle as explained by the great logician, Calvin

Axioms and Examples

Although Haskell's type system is very powerful, it is not quite good enough to prove everything that we will need. For this reason, we will need some axioms, or propositions that we assume without proof. We will use `admit` to denote that we take a proposotion to be true. In order to get this to typecheck, we define `admit` as follows:

```
admit :: assumption
admit = admit
```

`admit` is simply an infinite loop, but it can have any type! This is the same way that `undefined` is written in the Haskell Prelude.

We can now write down the Law of Excluded Middle:

$$\frac{}{P \vee \neg P}$$

Figure 5: Law of Excluded Middle

This states that some proposition P is either true or it is false. Haskell's type system cannot prove this, so we leave it as an admitted axiom:

```
excluded_middle :: p \/ Not p
excluded_middle = admit
```

A few more propositions are defined in `HW08.hs`. Try to understand them before beginning the exercises. In particular, there is a fully annotated proof of the Material Implication Theorem.

$$\frac{}{(P \to Q) \leftrightarrow (\neg P \vee Q)}$$

Figure 6: Material Implication

This theorem states that implication can be equivalently written as a disjunction.

Exercise 1 In this exercise you will prove Disjunctive Syllogism. In logic, we can write this theorem as follows:

$$\frac{P \vee Q \qquad \neg P}{Q}$$

Figure 7: Disjunctive Syllogism

Basically, it states that if one of P or Q is true, but we know that P is false, then Q must be true. The proof is relatively straighforward; you should do case analysis on the proposition $P \vee Q$. In the left case, you can derive a contradiction and in the right case, you simply know that Q is true.

```
disjunctive_syllogism :: (p \/ q) -> Not p -> q
```

Exercise 2 Prove the Composition Theorem:

$$\frac{(P \to Q) \vee (P \to R) \qquad P}{Q \vee R}$$

Figure 8: Composition

The proof of this theorem is again by case analysis on the disjunction hypothesis. The left case in the hypothesis corresponds to the

left case in the conclusion and the right case in the hypothesis corresponds to the right case in the conclusion.

Exercise 3 Prove the Transposition Theorem:

$$\frac{}{(P \to Q) \leftrightarrow (\neg Q \to \neg P)}$$

Figure 9: Transposition

The proof of Transposition has two parts; the forward direction and the backwards direction. In the forward direction, you are essentially proving:

$$\frac{\begin{array}{l} P \to Q \\ \neg Q \end{array}}{\neg P}$$

Figure 10: Forward Direction of Transposition

This is identical to a theorem that we already proved! The backwards direction takes a bit more work

$$\frac{\begin{array}{l} \neg Q \to \neg P \\ P \end{array}}{Q}$$

Figure 11: Backward Direction of Transposition

This can be proven using an application of Modus Tollens, but with a slight catch. You will have to get P into the form $\neg\neg P$ before you can use Modus Tollens and then you will have to remove the double negation at the end.

```
transposition :: (p -> q) <-> (Not q -> Not p)
```

Exercise 4 Prove one of De Morgan's Laws

$$\frac{}{\neg(P \vee Q) \leftrightarrow (\neg P \wedge \neg Q)}$$

Figure 12: De Morgan's Law

Again, you will have to prove the forward and backward directions. Recall that $\neg P$ is defined as $P \to \bot$ and you can construct a proposition of type $P \vee Q$ with *either* a P or a Q.

```
de_morgan :: Not (p \/ q) <-> (Not p /\ Not q)
```

Natural Numbers

The natural numbers, often denoted as the set $\mathbb{N} = \{0, 1, 2, \ldots\}$, are simply the non-negative integers. In class we defined natural numbers and used them to encode the length of a vector in its type. In

this way, we promoted the data type `Nat` to be a kind and the constructors `S` and `O` were promoted to types. This is great for encoding numbers at the *type-level*, however we would also like to encode numerical values at the *expression-level*. In particular, we want numbers whose types are equivalent to their values. We can achieve this using the following Generalized Algebraic Data Type:

```
data Forall n where
  Zero :: Forall O
  Succ :: Forall n -> Forall (S n)
```

This is similar to the GADT that we defined for vectors, except that `Succ` (the equivalent of `Cons`) does not take in a value, only a size. We name this type `Forall` because it is notationally convenient in proofs, however for all intents and purposes, a value of type `Forall n` is a number.

Before we can write down some proofs, we also need to define a notion of equality. Equality is defined using a GADT:

```
data (n :: Nat) == (m :: Nat) where
  Refl :: n == n
```

This defines equality for two types, `n` and `m`, of kind `Nat`. Notice that there is only one constructor and it has type `n == n`. This is because the only way for two `Nat`s to be equal is if they are the same. In other words, the equality relation is *reflexive*.

We also define the less than relation as a GADT with two constructors:

```
data n < m where
  LT_Base :: O < S n
  LT_Rec  :: n < m -> S n < S m
```

Less than is defined inductively. The base case states that zero is less than any number that is the successor of some other number. The recursive case is constructed using a witness that `n` is less than `m` and produces a proof that the successor of `n` is less than the successor of `m`.

Before you start doing the exercises in this section, read through the annotated proof of the "Not Equal implies Greater Than or Less Than" theorem. This theorem states that:

$$\forall n \in \mathbb{N},\ \forall m \in \mathbb{N},\ n \neq m \rightarrow (n < m \vee n > m)$$

and in Haskell it is written as:

```
neq_gt_lt :: Forall n -> Forall m ->
             n /= m -> (n < m) \/ (n > m)
```

Figure 13: Portrait of the Proof General. The Proof General verifies proofs written in several langauages including Coq, LEGO, and Isabelle. Unfortunately, he does not work in Haskell.

The reason why we need the `Forall` type is shown here. We have to do induction on `n` and `m` in order to complete the proof. Since the number `Forall n` has the same type-level numerical value as the `n` that shows up elsewhere in the theorem, this gives us a way to manipulate the value of the type at the expression level. In particular, it allows us to specify the values of `n` and `m` in recursive calls.

Exercise 5 Notice how the theorem `0 + n == n` is trivial to prove whereas `n + 0 == n` requires an inductive proof. The reason for this is that `0 + n == n` follows directly from the definition of the addition type family.

Look at the proof of `n_plus_0`. The conclusion comes after a case match where we match `Refl` and then return `Refl`. This seems silly, why couldn't we skip the match and just return `n_plus_0 n`?

The reason is that the two `Refl` proof objects have different types, however the Haskell compiler needs to see that the result of the recursive call is a `Refl` before it can verify that the type of the returned `Refl` is correct.

Your task is to prove a similar theorem. The structure of this proof is almost identical to `n_plus_0`.

```
add_zero :: Forall n -> O + n == n + O
```

Exercise 6 Now, prove the property that $\forall n,\ n < n + 1$. This theorem also has a straightforward proof by induction on n.

```
n_lt_sn :: Forall n -> n < S n
```

Exercise 7 The last two exercises will use new relations for even an odd numbers. Similar to the less than relation, these relations are defined inductively. The even relation states that zero is even, and if `n` is even, then `S (S n)` is even. The odd relation is the same except that it has a base case on 1 instead of 0.

Provided for you is a proof that the successor of an even number is odd. Unlike the previous proofs that were done by induction on `n`, this proof is by induction on the even judgement. Your job is to prove that the successor of an odd number is even.

```
odd_plus_one :: Odd n -> Even (S n)
```

Exercise 8 Finally, prove that for any number `n`, `n + n` is even. This theorem requires a proof by induction on `n`. You may find the `succ_sum` lemma helpful!

```
double_even :: Forall n -> Even (n + n)
```