1. Simulating with Polymorphic Types (25 pts)

1.1. Simulating Pairs

One can get a lot of mileage out of just function and polymorphic types. Consider the following type definition (for $\tau_1$ and $\tau_2$ are any two types not containing $\alpha$):

$$\text{prod}_{\tau_1, \tau_2} = \forall \alpha. (\tau_1 \rightarrow \tau_2 \rightarrow \alpha) \rightarrow \alpha$$

Think about what kind of values of type prod$_{\tau_1, \tau_2}$ there could be. Since it is a $\forall$-type, a value of this type must start with $\Lambda \alpha$. and then produce something of type $(\tau_1 \rightarrow \tau_2 \rightarrow \alpha) \rightarrow \alpha$ for an arbitrary $\alpha$. But the only way we have of producing $\alpha$-typed data is the function $\tau_1 \rightarrow \tau_2 \rightarrow \alpha$ that we’ve received as an argument. Thus we must call this function and pass it something of type $\tau_1$ and something else of type $\tau_2$. So every value prod$_{\tau_1, \tau_2}$ is like a pair of an expression of type $\tau_1$ and an expression of type $\tau_2$.

**Question 1** (5 points). Write in MinML the following functions:

- $\text{pair} : \forall \beta_1. \forall \beta_2. \beta_1 \rightarrow \beta_2 \rightarrow \text{prod}_{\beta_1, \beta_2}$
- $\text{fst} : \forall \beta_1. \forall \beta_2. \text{prod}_{\beta_1, \beta_2} \rightarrow \beta_1$
- $\text{snd} : \forall \beta_1. \forall \beta_2. \text{prod}_{\beta_1, \beta_2} \rightarrow \beta_2$
The function `pair` should take a value of type $\beta_1$ and one of $\beta_2$ and produce something of type $\forall \alpha. (\beta_1 \rightarrow \beta_2 \rightarrow \alpha) \rightarrow \alpha$ that represents a pair of the two inputs as suggestively described above. Then `fst` and `snd` should accordingly return the appropriate components of the pair.

1.3. Simulating Existential Types

Similarly we can simulate existential types using just polymorphic types and functions. Here is the definition:

$\text{exist}_{\beta.\tau} = \forall \alpha. (\forall \beta. (\tau \rightarrow \alpha)) \rightarrow \alpha$

In both the left and right hand of the equation, $\tau$ is a type that may refer to the variable $\beta$ (but not $\alpha$). The reasoning here is that to make something of type $\text{exist}_{\beta.\tau}$, we must make something that is of type $(\forall \beta. \tau \rightarrow \alpha) \rightarrow \alpha$ for any arbitrary $\alpha$. The only way we can write a function is by taking in an argument, say $v$, of type $\forall \beta. \tau \rightarrow \alpha$ and then apply it to something. Since $v$ accepts something of type $\tau$ for any type $\beta$ (remember that $\beta$ may occur in $\tau$!), what we are required to provide to $v$ is something of type $\tau$ for some $\beta$. This notion of ‘some’ is exactly what is meant by an existential quantifier $\exists$.

To summarize, to construct a value of type $\text{exist}_{\beta.\tau}$, we must construct something of type $\exists \beta.\tau$.

Question 2 (10 points). Describe how to translate MinML programs that use existential types into programs that instead use the type $\text{exist}_{\beta.\tau}$. You will need to at least sketch out a recursive translation that replaces every instance of `pack` and `open` with expressions not involving these constructs. Could we have instead simply written functions to replace `pack` and `open` as we did with `fst` and `snd` and `pair` above? Why or why not? (Hint: what type would our `pack` replacement have?)

1.3. Simulating a Mystery Type

Similar to the above encodings, the following type is ‘morally equivalent’ to one of the familiar type constructors we’ve considered in class already.

$mystery_{\tau_1.\tau_2} = \forall \alpha. (\tau_1 \rightarrow \alpha) \rightarrow (\tau_2 \rightarrow \alpha) \rightarrow \alpha$

Question 3 (5 points). Which is it? What are the constructors and deconstructors for this type constructor? Give their types, and implement them in the same general way as part 1.1.
1.4. Evaluation

While these encodings may seem intuitively plausible, the proof of their correctness must proceed carefully.

**Question 4** (5 points). Find a counterexample to the following first attempt at a correctness theorem:

**Theorem 1.** Let $e_1$ be an expression in MinML with product types, and let $e_2$ be the translation of $e_1$ obtained by replacing every instance of $\tau_1 \times \tau_2$ with $\text{prod}_{\tau_1, \tau_2}$, and all of the pair constructors and deconstructors with the functions you wrote in part 1.1.

Assuming this, if $e_1 \rightarrow e_1'$, then $e_2 \rightarrow e_2'$ for expressions $e_1'$ and $e_2'$ such that $e_2'$ is similarly the translation of $e_1'$.

2. Existential and Recursive Types (25 pts)

Consider the following datatype of bitstrings:

```
datatype bits = Empty
  | One of bits
  | Zero of bits
```

Just to be unambiguous about left-to-right order, the bitstring 1011 is represented as `One(Zero(One(One(Empty))))`.

**Question 5** (5 points). Express the `bits` datatype as a recursive (i.e. $\mu$) type.

**Question 6** (5 points). Write a function `or : bits * bits → bits` over this recursive type to compute the binary OR of two arguments. If one argument is not the same length as the other, behave as if you extended the shorter one on the right with zeros. For example, the OR of 101 and 00101 is 10101.

**Question 7** (5 points). Express the `bits` datatype as an abstract type, (i.e. using $\exists$) exposing the three constructors and the deconstructor.

**Question 8** (10 points). Rewrite `or` to work on this abstract type.