Task 1 (L20.1, 30 points) A lazy record is a generalization of a lazy pair where each alternative has a different label $i$. For example, potentially infinite streams $\text{stream } \alpha$ of elements of some type $\alpha$ may be defined as

\[
\text{stream } \alpha \cong (\text{hd} : \alpha) \& (\text{tl} : \text{stream } \alpha)
\]

As an example of the general syntax $\langle i \Rightarrow e \rangle_i \in I$ for a lazy record with the fields in the finite index set $I$, we show how to define a stream of just 0s (omitting the standard definitions of zero and succ):

\[
\begin{align*}
\text{nat} & \cong (z : 1) + (s : \text{nat}) \\
\text{zero} & : \text{nat} \\
\text{succ} & : \text{nat} \rightarrow \text{nat} \\
\text{zeros} & : \text{stream } \text{nat} \\
\text{zeros} & = \text{fold } \langle \text{hd} \Rightarrow \text{zero}, \text{tl} \Rightarrow \text{zeros} \rangle
\end{align*}
\]

In fully explicit form, the definition of zeros would be a fixed point:

\[
\text{zeros} = \text{fix } f. \text{fold } \langle \text{hd} \Rightarrow \text{zero}, \text{tl} \Rightarrow f \rangle
\]

but we prefer the first form where the recursion is implicit. This definition terminates because the record with field hd and tl is lazy. We select an element of a lazy record $e$ by writing $e \cdot j$ for a label $j$ (which is just the postfix version of the injection into a sum $j \cdot e$). As an example, the following function adds 1 to every element of the given stream.

\[
\begin{align*}
\text{succs} & : \text{stream } \text{nat} \rightarrow \text{stream } \text{nat} \\
\text{succs} & = \lambda s. \text{fold } \langle \text{hd} \Rightarrow \text{succ } ((\text{unfold } s) \cdot \text{hd}), \text{tl} \Rightarrow \text{succs } ((\text{unfold } s) \cdot \text{tl}) \rangle \\
\text{ones} & = \text{succs } \text{zeros}
\end{align*}
\]

Write functions satisfying the following specifications:

1. \text{up from} : \text{nat} \rightarrow \text{stream } \text{nat} where \text{up from } n generates the stream $n, n + 1, n + 2, \ldots$
2. \( \forall \alpha. \text{stream } \alpha \to \text{stream } \alpha \to \text{stream } \alpha \) which alternates the elements from the two streams, starting with the first element of the first stream.

\[
\text{alt} : \quad \forall \alpha. \text{stream } \alpha \to \text{stream } \alpha \to \text{stream } \alpha \\
\text{alt} = \quad \Lambda \alpha. \lambda s. \lambda t. \text{fold} \langle | \text{hd} \Rightarrow (\text{unfold } s).\text{hd}, \text{tl} \Rightarrow \text{alt} [\alpha] t (\text{unfold } s).\text{tl} | \rangle
\]

3. \( \forall \alpha. (\alpha \to \text{bool}) \to \text{stream } \alpha \to \text{stream } \alpha \) which returns the stream with just those elements of the input stream that satisfy the given predicate.

The first alternative solution is very lazy (nothing is read from the input stream unless the output stream is accessed) but requires the condition to be evaluated twice in case both head and tail of the resulting stream are accessed.

\[
\text{filter} : \quad \forall \alpha. (\alpha \to \text{bool}) \to \text{stream } \alpha \to \text{stream } \alpha \\
\text{filter} = \quad \Lambda \alpha. \lambda p. \lambda s. \text{let } x = (\text{unfold } s).\text{hd} \text{ in case } p \ x (\text{true } \Rightarrow x | \text{false } \Rightarrow (\text{unfold } (\text{filter} [\alpha] p ((\text{unfold } s).\text{tl}))).\text{hd}), \text{tl } \Rightarrow \text{case } p ((\text{unfold } (\text{filter} [\alpha] p ((\text{unfold } s).\text{tl}))).\text{tl}) | \text{false } \Rightarrow (\text{unfold } (\text{filter} [\alpha] p ((\text{unfold } s).\text{tl}))).\text{tl})
\]

Most of you gave an alternative definition for this function \((\text{filter'})\) which is slightly more eager:

\[
\text{filter'} : \quad \forall \alpha. (\alpha \to \text{bool}) \to \text{stream } \alpha \to \text{stream } \alpha \\
\text{filter'} = \quad \Lambda \alpha. \lambda p. \lambda s. \text{let } x = (\text{unfold } s).\text{hd} \text{ in case } p \ x (\text{true } \Rightarrow \text{fold} \langle \text{hd} \Rightarrow x, \text{tl } \Rightarrow \text{filter'} [\alpha] p ((\text{unfold } s).\text{tl}) | \text{false } \Rightarrow \text{filter'} [\alpha] p ((\text{unfold } s).\text{tl})
\]

In this alternative definition if computing the head of a given stream \(((\text{unfold } s).\text{hd})\) doesn’t terminate for some reason, the entire function doesn’t terminate. In contrast having the first definition, in the same situation, \(\text{filter}\) terminates until it is asked for the head or the tail.

4. \( \forall \alpha. \forall \beta. (\alpha \to \beta) \to (\text{stream } \alpha \to \text{stream } \beta) \) which returns a stream with the result of applying the given function to every element of the input stream.
map : \\forall \alpha. \forall \beta. (\alpha \to \beta) \to (\text{stream } \alpha \to \text{stream } \beta)
map = \Lambda \alpha. \Lambda \beta. \lambda f. \lambda s. \text{fold } \langle \text{hd } \Rightarrow f (\text{unfold } s). \text{hd}, \text{tl } \Rightarrow \text{map } \[ \alpha \] \[ \beta \] f (\text{unfold } s). \text{tl} \rangle

5. diag : \forall \alpha. \text{stream } (\text{stream } \alpha) \to \text{stream } \alpha which returns a stream consisting of the first element of the first stream, the second element of the second stream, the third element of the third stream, etc.

We first define a auxiliary function aux : \forall \alpha. \text{stream } (\text{stream } \alpha) \to \text{stream } (\text{stream } \alpha) that deletes the head of every stream:

aux : \forall \alpha. \text{stream } (\text{stream } \alpha) \to \text{stream } (\text{stream } \alpha)
aux = \Lambda \alpha. \lambda t. \text{fold } \langle \text{hd } \Rightarrow \text{unfold } ((\text{unfold } t). \text{hd}). \text{tl}, \text{tl } \Rightarrow \text{aux } \[ \alpha \] (\text{unfold } t). \text{tl} \rangle

Now we can define the diagonal function:

diag : \forall \alpha. \text{stream } (\text{stream } \alpha) \to \text{stream } \alpha
diag = \Lambda \alpha. \lambda t. \text{fold } \langle \text{hd } \Rightarrow \text{unfold } ((\text{unfold } t). \text{hd}). \text{hd}, \text{tl } \Rightarrow \text{diag } \[ \alpha \] aux (\text{unfold } t). \text{tl} \rangle

You may use earlier functions in the definition of later ones. To avoid some recomputation, you may use the syntactic sugar of let \( x = e \) in \( e' \) to stand for \( (\lambda x. e') e \).

Your functions should be such that only as much of the output stream is computed as necessary to obtain a value of type \text{stream } \alpha but not the components contained in the lazy record. For example, the definition of \text{succs}' below would be still terminating, but slightly too eager (for example, we may never access the element at the head of the resulting stream), while the second \text{succs}'' would not even be terminating any more.

\text{succs}' = \lambda s. \text{let } x = \text{succ} ((\text{unfold } s) \cdot \text{hd})
\quad \text{in } \text{fold } \langle \text{hd } \Rightarrow x, \text{tl } \Rightarrow \text{succs}' ((\text{unfold } s) \cdot \text{tl}) \rangle

\text{succs}'' = \lambda s. \text{let } s' = \text{succs}'' ((\text{unfold } s) \cdot \text{tl})
\quad \text{in } \text{fold } \langle \text{hd } \Rightarrow \text{succ} ((\text{unfold } s) \cdot \text{hd}), \text{tl } \Rightarrow s' \rangle

Task 2 (L20.2, 30 points) For lazy records as introduced in Task 1 we introduce the following syntax in our language of expressions:

\begin{align*}
\text{Types} \quad &::= \ldots \mid \tau \in \mathcal{I} (i : \tau_i) \\
\text{Expressions} \quad &::= \ldots \mid e_i \in \mathcal{I} (i : e) \cdot j
\end{align*}

1. Give the typing rules and the dynamics (stepping rules) for the new constructs.
2. Extend the translation \[[\langle i \Rightarrow e_i \rangle_{i \in I} \mapsto d]\) to encompass the new constructs. Your process syntax should expose the duality between eager sums and lazy records.

\[
\begin{align*}
\langle i \Rightarrow e_i \rangle_{i \in I} \mapsto d &= \text{case } d^W (i)(x) \Rightarrow [e_i]_{i \in I} \\
\langle e.i \rangle_{i \in I} \mapsto d &= d_1 \leftarrow [e]_{d_1} d^R.j(d). \quad \text{(Put the jth element of } d_1 \text{ in } d.)
\end{align*}
\]

Just as we had for sums, passing value to continuation results in

\[j(d) \triangleright (i(y) \Rightarrow P_i)_{i \in I} = ([d/y]P_j)\]

3. Extend the transition rules of the store-based dynamics to the new constructs. The translated form may permit more parallelism than the original expression evaluation, but when scheduled sequentially they should have the same behavior (which you do not need to prove).

4. Show the typing rules for the new process constructs.

\[
\begin{align*}
j \in I & \quad x : \&_{i \in I} (i : \tau_i) \in \Gamma \\
\Gamma \vdash x^R.j(y) :: (y : \tau_j) & \quad \text{lazy/read} \\
\forall i \in I & \quad \Gamma \vdash P_i :: (x : \tau_i) \\
\Gamma \vdash \text{case } y^W (i)(x) \Rightarrow P_i)_{i \in I} :: (y : \&_{i \in I} (i : \tau_i)) & \quad \text{lazy/write}
\end{align*}
\]