Assignment 9
Lazy Records

15-814: Types and Programming Languages
Frank Pfenning

Due Tuesday, November 26, 2019

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 *stream* α of elements of some type α may be defined as

$$\textit{stream}\ \alpha \quad \cong \quad (\mathtt{hd} : \alpha)\ \&\ (\mathtt{tl} : \textit{stream}\ \alpha)$$

As an example of the general syntax $\langle i \Rightarrow e_i \rangle_{i \in I}$ 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*):

$$\begin{array}{lcl}
\textit{nat} & \cong & (\mathtt{z} : 1) + (\mathtt{s} : nat) \\
\textit{zero} & : & \textit{nat} \\
\textit{succ} & : & \textit{nat} \to \textit{nat} \\
\textit{zeros} & : & \textit{stream nat} \\
\textit{zeros} & = & \mathsf{fold}\ \langle \mathtt{hd} \Rightarrow \textit{zero}, \mathtt{tl} \Rightarrow \textit{zeros} \rangle
\end{array}$$

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

$$\textit{zeros} = \mathsf{fix}\ f.\ \mathsf{fold}\ \langle \mathtt{hd} \Rightarrow \textit{zero}, \mathtt{tl} \Rightarrow f \rangle$$

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 elements of the given stream.

$$\begin{array}{lcl}
\textit{succs} & : & \textit{stream nat} \to \textit{stream nat} \\
\textit{succs} & = & \lambda s.\ \mathsf{fold}\ \langle \mathtt{hd} \Rightarrow \textit{succ}\ ((\mathsf{unfold}\ s) \cdot \mathtt{hd}), \mathtt{tl} \Rightarrow \textit{succs}\ ((\mathsf{unfold}\ s) \cdot \mathtt{tl}) \rangle \\
\textit{ones} & = & \textit{succs zeros}
\end{array}$$

Write functions satisfying the following specifications:

1. *up_from* : *nat* $\to$ *stream nat* where *up_from* n generates the stream $n, n+1, n+2, \ldots$.
2. *alt* : $\forall \alpha.\ \textit{stream}\ \alpha \to \textit{stream}\ \alpha \to \textit{stream}\ \alpha$ which alternates the elements from the two streams, starting with the first element of the first stream.

3. *filter* : $\forall\alpha.\,(\alpha \to \mathit{bool}) \to \mathit{stream}\ \alpha \to \mathit{stream}\ \alpha$ which returns the stream with just those elements of the input stream that satisfy the given predicate.

4. *map* : $\forall\alpha.\forall\beta.\,(\alpha \to \beta) \to (\mathit{stream}\ \alpha \to \mathit{stream}\ \beta)$ which returns a stream with the result of applying the given function to every element of the input stream.

5. *diag* : $\forall\alpha.\,\mathit{stream}\ (\mathit{stream}\ \alpha) \to \mathit{stream}\ \alpha$ 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.

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 *stream* α but not the components contained in the lazy record. For example, the definition of *succs*′ below would be still terminating, but slighty too eager (for example, we may never access the element at the head of the resulting stream) , while the second succs″ would not even be terminating any more.

$$\begin{array}{l}\mathit{succs}' = \lambda s.\ \mathsf{let}\ x = \mathit{succ}\ ((\mathsf{unfold}\ s) \cdot \mathtt{hd}) \\ \qquad\qquad \mathsf{in\ fold}\ \langle\!|\mathtt{hd} \Rightarrow x, \mathtt{tl} \Rightarrow \mathit{succs}'\ ((\mathsf{unfold}\ s) \cdot \mathtt{tl})\end{array}$$

$$\begin{array}{l}\mathit{succs}'' = \lambda s.\ \mathsf{let}\ s' = \mathit{succs}''\ ((\mathsf{unfold}\ s) \cdot \mathtt{tl}) \\ \qquad\qquad \mathsf{in\ fold}\ \langle\!|\mathtt{hd} \Rightarrow \mathit{succ}\ ((\mathsf{unfold}\ s) \cdot \mathtt{hd}), \mathtt{tl} \Rightarrow s'|\!\rangle\end{array}$$

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

$$\begin{array}{lll}\text{Types} & ::= & \ldots \mid \&_{i \in I}(i : \tau_i) \\ \text{Expressions} & ::= & \ldots \mid \langle\!|i \Rightarrow e_i|\!\rangle_{i \in I} \mid e \cdot j\end{array}$$

1. Give the typing rules and the dynamics (stepping rules) for the new constructs.

2. Extend the translation $[\![e]\!]\,d$ to encompass the new constructs. Your process syntax should expose the duality between eager sums and lazy records.

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.