Lecture Notes on Negative Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 20
Tuesday, November 12, 2019

1 Introduction

We continue the investigation of shared memory concurrency by adding *negative types*. In our language so far they are functions $\tau \to \sigma$, lazy pairs $\tau \,\&\, \sigma$, and universal types $\forall\alpha.\,\tau$.

2 Review of Positives

We review the types so far, with a twist: we annotate every address that we write to with a superscript^W and every address we read from with a superscript^R.

$$
\begin{array}{llllll}
\text{Processes} & P & ::= & x \leftarrow P \,;\, Q & & \text{allocate/spawn} \\
 & & \mid & x^W \leftarrow y^R & & \text{copy} \\
 & & \mid & x^W.\langle\,\rangle & \mid \mathsf{case}\ x^R\ (\langle\,\rangle \Rightarrow P) & (1) \\
 & & \mid & x^W.\langle y, z\rangle & \mid \mathsf{case}\ x^R\ (\langle y, z\rangle \Rightarrow P) & (\times) \\
 & & \mid & x^W.j(y) & \mid \mathsf{case}\ x^R\ (i(y) \Rightarrow P_i)_{i\in I} & (+) \\
 & & \mid & x^W.\mathsf{fold}(y) & \mid \mathsf{case}\ x^R\ (\mathsf{fold}(y) \Rightarrow P) & (\rho) \\
\text{Cell Contents} & W & ::= & \langle\,\rangle \mid \langle d_1, d_2\rangle \mid j(d) \mid \mathsf{fold}(d) & & \\
\text{Configurations} & \mathcal{C} & ::= & \cdot \mid \mathcal{C}_1, \mathcal{C}_2 \mid \mathsf{proc}\ d\ P, \mathsf{cell}\ d\ \_ \mid \mathsf{!cell}\ c\ W & &
\end{array}
$$

The configurations are unordered and we think of "," as an associative and commutative operator with unit "$\cdot$". Since we have changed our notation a few times, we summarize the translation and the transition rules. We also throw in sums, but we do not explicitly write out the case for recursive types, which follows the pattern established by the other cases.

$$[\![x]\!]\,d = d^W \leftarrow x^R$$

$$\begin{aligned}
[\![\langle\,\rangle]\!]\,d &= d^W.\langle\,\rangle \\
[\![\mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e')]\!]\,d &= d_1 \leftarrow [\![e]\!]\,d_1\ ; \\
&\quad\ \mathsf{case}\ d_1^R\ (\langle\,\rangle \Rightarrow [\![e']\!]\,d)
\end{aligned}$$

$$\begin{aligned}
[\![\langle e_1, e_2\rangle]\!]\,d &= d_1 \leftarrow [\![e_1]\!]\,d_1\ ; \\
&\quad\ d_2 \leftarrow [\![e_2]\!]\,d_2\ ; \\
&\quad\ d^W.\langle d_1, d_2\rangle
\end{aligned}$$

$$\begin{aligned}
[\![\mathsf{case}\ e\ (\langle x_1, x_2\rangle \Rightarrow e')]\!]\,d &= d_1 \leftarrow [\![e]\!]\,d_1\ ; \\
&\quad\ \mathsf{case}\ d_1^R\ (\langle x_1, x_2\rangle \Rightarrow [\![e']\!]\,d)
\end{aligned}$$

$$\begin{aligned}
[\![j \cdot e]\!]\,d &= d_1 \leftarrow [\![e]\!]\,d_1\ ; \\
&\quad\ d^W.j(d_1)
\end{aligned}$$

$$\begin{aligned}
[\![\mathsf{case}\ e\ (i \cdot x \Rightarrow e_i)_{i \in I}]\!]\,d &= d_1 \leftarrow [\![e]\!]\,d_1\ ; \\
&\quad\ \mathsf{case}\ d_1^R\ (i(x) \Rightarrow [\![e_i]\!]\,d)_{i \in I}
\end{aligned}$$

And the computation rules for *configurations*:

$$\mathsf{proc}\ d'\ (x \leftarrow P\ ;\ Q) \mapsto \mathsf{proc}\ d\ ([d/x]P), \mathsf{cell}\ d\ \_, \mathsf{proc}\ d'\ ([d/x]Q) \quad (d\ \text{fresh}) \qquad \text{(alloc/spawn)}$$

$$!\mathsf{cell}\ c\ W, \mathsf{proc}\ d\ (d \leftarrow c), \mathsf{cell}\ d\ \_ \mapsto\ !\mathsf{cell}\ d\ W \qquad \text{(copy)}$$

$$\begin{aligned}
&\mathsf{proc}\ d\ (d.\langle\,\rangle), \mathsf{cell}\ d\ \_ \mapsto\ !\mathsf{cell}\ d\ \langle\,\rangle && (1R^0) \\
&!\mathsf{cell}\ c\ \langle\,\rangle, \mathsf{proc}\ d\ (\mathsf{case}\ c\ (\langle\,\rangle \Rightarrow P)) \mapsto \mathsf{proc}\ d\ P && (1L)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{proc}\ d\ (d.\langle c_1, c_2\rangle), \mathsf{cell}\ d\ \_ \mapsto\ !\mathsf{cell}\ d\ \langle c_1, c_2\rangle && (\times R^0) \\
&!\mathsf{cell}\ c\ \langle c_1, c_2\rangle, \mathsf{proc}\ d\ (\mathsf{case}\ c\ (\langle x_1, x_2\rangle \Rightarrow P)) \mapsto \mathsf{proc}\ d\ ([c_1/x_1, c_2/x_2]P) && (\times L)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{proc}\ d\ (d.j(c)), \mathsf{cell}\ d\ \_ \mapsto\ !\mathsf{cell}\ d\ (j(c)) && (+R^0) \\
&!\mathsf{cell}\ c\ (j(c_1)), \mathsf{proc}\ d\ (\mathsf{case}\ c\ (i \cdot x \Rightarrow P_i)_{i \in I} \mapsto \mathsf{proc}\ d\ ([c_1/x]P_j) && (+L)
\end{aligned}$$

3 Functions

As the first negative type we consider function $\tau \to \sigma$. How do we translate an abstraction $\lambda x.\, e$? The translation must actually take *two* arguments: one is the original argument x, the other is the destination where the result of the functional call should be written to. And the process $[\![\lambda x.\, e]\!]\,d$ must write the translation of the function to destination d.

Before we settle on the syntax for this, consider how to translate function application.

$$\begin{aligned}
[\![e_1\, e_2]\!]\, d = {} & d_1 \leftarrow [\![e_1]\!]\, d_1 \;; \\
& d_2 \leftarrow [\![e_2]\!]\, d_2 \;; \\
& \boxed{}
\end{aligned}$$

How should we complete this translation?

We know that after $[\![e_1]\!]\, d_1$ has completed the cell e_1 will contain *a function of two arguments*. The *first* argument is the original argument x, which we find in d_2 after $[\![e_2]\!]\, d_2$ has completed. The *second* argument is the destination for the result of the functional application, which is d. So we get:

$$\begin{aligned}
[\![e_1\, e_2]\!]\, d = {} & d_1 \leftarrow [\![e_1]\!]\, d_1 \;; \\
& d_2 \leftarrow [\![e_2]\!]\, d_2 \;; \\
& d_1^R.\langle d_2, d\rangle
\end{aligned}$$

This looks just like eager pairs, except that we *read* from d_1 instead of writing to it. To retain the analogy, we write the translation of a function using case, but *writing* the (single) branch of the case expression to memory.

$$[\![\lambda x.\, e]\!]\, d = \mathsf{case}\ d^W\ (\langle x, y\rangle \Rightarrow [\![e]\!]\, y)$$

The transition rules for these new constructs just formalize the explanation.

$$\begin{array}{lr}
\mathsf{proc}\ d\ (\mathsf{case}\ d\ (\langle x,y\rangle \Rightarrow P)), \mathsf{cell}\ d\ \_ \mapsto\ !\mathsf{cell}\ d\ (\langle x,y\rangle \Rightarrow P) & ({\to}R) \\
!\mathsf{cell}\ c\ (\langle x,y\rangle \Rightarrow P), \mathsf{proc}\ d\ (c.\langle c_1, c_2\rangle) \mapsto \mathsf{proc}\ d\ ([c_1/x, c_2/y]P) & ({\to}L^0)
\end{array}$$

As an example, we consider the expression $(\lambda x.\, x)\,\langle\,\rangle$.

$$\begin{aligned}
[\![(\lambda x.\, x)\,\langle\,\rangle]\!]\, d_0 &= d_1 \leftarrow [\![\lambda x.\, x]\!]\, d_1 \;; \\
&\quad\ d_2 \leftarrow [\![\langle\,\rangle]\!]\, d_2 \;; \\
&\quad\ d_1^R.\langle d_2, d_0\rangle \\
&= d_1 \leftarrow \mathsf{case}\ d_1^W\ (\langle x, y\rangle \Rightarrow [\![x]\!]\, y) \;; \\
&\quad\ d_2 \leftarrow d_2^W.\langle\,\rangle \;; \\
&\quad\ d_1^R.\langle d_2, d_0\rangle \\
&= d_1 \leftarrow \mathsf{case}\ d_1^W\ (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R) \;; \\
&\quad\ d_2 \leftarrow d_2^W.\langle\,\rangle \;; \\
&\quad\ d_1^R.\langle d_2, d_0\rangle
\end{aligned}$$

Let's execute the final process from with the initial destination d_0.

$$
\begin{array}{ll}
 & \mathsf{proc}\ d_0\ (d_1 \leftarrow \mathsf{case}\ d_1^W\ (\ldots)\ ;\ d_2 \leftarrow d_2^W.\langle\,\rangle\ ;\ \ldots), \mathsf{cell}\ d_0\ \_ \\
\mapsto & \mathsf{proc}\ d_1\ (\mathsf{case}\ d_1^W\ (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R), \mathsf{cell}\ d_1\ \_, \\
 & \mathsf{proc}\ d_0\ (d_2 \leftarrow d_2^W.\langle\,\rangle\ ;\ d_1^R.\langle d_2, d_0\rangle), \mathsf{cell}\ d_0\ \_ \\
\mapsto^2 & !\mathsf{cell}\ d_1\ (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R), \\
 & \mathsf{proc}\ d_2\ (d_2^W.\langle\,\rangle), \mathsf{cell}\ d_2\ \_, \\
 & \mathsf{proc}\ d_0\ (d_1^R.\langle d_2, d_0\rangle), \mathsf{cell}\ d_0\ \_ \\
\mapsto^2 & !\mathsf{cell}\ d_1\ (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R), \\
 & !\mathsf{cell}\ d_2\ \langle\,\rangle \\
 & \mathsf{proc}\ d_0\ (d_0^W \leftarrow d_2^R), \mathsf{cell}\ d_0\ \_ \qquad\qquad (\text{from } [d_2/x, d_0/y](y^W \leftarrow x^R)) \\
\mapsto & !\mathsf{cell}\ d_1\ (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R), \\
 & !\mathsf{cell}\ d_2\ \langle\,\rangle \\
 & !\mathsf{cell}\ d_0\ \langle\,\rangle
\end{array}
$$

In the final state we have cell d_0 holding the final result $\langle\,\rangle$, which is indeed the result of evaluating $(\lambda x.\, x)\ \langle\,\rangle$. We also have some newly allocated intermediate destinations d_1 and d_2 that are preserved, but could be garbage collected if we only retain the cells that are reachable from the initial destination d_0 which now holds the final value.

4 Store Revisited

In our table of process expression, two things stand out. One is that functions are exactly like pairs, except that the role of reads and writes are reversed. The other is that a cell may now contain something of the form $(\langle y, z\rangle \Rightarrow P)$.

Processes	P	$::=$	$x \leftarrow P\ ;\ Q$		allocate/spawn
		$\mid$	$x^W \leftarrow y^R$		copy
		$\mid$	$x^W.\langle\,\rangle$	$\mid \mathsf{case}\ x^R\ (\langle\,\rangle \Rightarrow P)$	(1)
		$\mid$	$x^W.\langle y, z\rangle$	$\mid \mathsf{case}\ x^R\ (\langle y, z\rangle \Rightarrow P)$	$(\times)$
		$\mid$	$x^W.j(y)$	$\mid \mathsf{case}\ x^R\ (i(y) \Rightarrow P_i)_{i \in I}$	$(+)$
		$\mid$	$x^W.\mathsf{fold}(y)$	$\mid \mathsf{case}\ x^R\ (\mathsf{fold}(y) \Rightarrow P)$	(ρ)
		$\mid$	$x^R.\langle y, z\rangle$	$\mid \mathsf{case}\ x^W\ (\langle y, z\rangle \Rightarrow P)$	$(\rightarrow)$
Cell Contents	W	$::=$	$\langle\,\rangle \mid \langle d_1, d_2\rangle \mid j(d) \mid \mathsf{fold}(d)$		
		$\mid$	$(\langle x, y\rangle \Rightarrow P)$		
Configurations	$\mathcal{C}$	$::=$	$\cdot \mid \mathcal{C}_1, \mathcal{C}_2 \mid \mathsf{proc}\ d\ P, \mathsf{cell}\ d\ \_ \mid !\mathsf{cell}\ c\ W$		

There is now a legitimate concern that the contents of cells in memory is no longer "small", because a program P could be of arbitrary size. We call expressions following "case x" *continuations* K (which are different from the continuations in the K Machine). A continuation will be actually implemented as a *closure*, that is, a pair consisting of an environment and the address of code to be executed. The translation to get us to this form is called *closure conversion*, which we might discuss in a future lecture. For now, we are content with the observation that, yes, we are violating a basic principle of fixed-size storage and that it can be mitigated (but is not completely solved) through the introduction of closures.

In our example of $(\lambda x.\, x)\ \langle\,\rangle$ the continuation has the form $(\langle x, y\rangle \Rightarrow y^W \leftarrow x^R)$ which is a closed expression in that has no free variables. This can be directly compiled to a function that takes two addresses x and y and writes the contents of x into y. So at least in this special case the contents of the cell d_1 could simply be the address of this piece of code.

The symmetry between eager pairs (positive) and functions (negative) stems from the property that in logic we have $A \vdash B \supset C$ if and only if $A \times B \vdash C$ (where $\times$ is a particular form of conjunction). Or, we can chalk it up to the isomorphism $\tau \to (\sigma \to \rho) \cong (\tau \times \sigma) \to \rho$: an arrow on the right behaves like a product on the left.

One can ask if similarly symmetric constructors exists for 1 and $+$ and the answer is yes. It turns out that lazy pairs are symmetric to sums and there is a type $\bot$ that is symmetric to 1 (see Exercises 2 and 3). There may even be a lazy analogue of recursive types that exhibits the same kind of symmetry and maybe useful to model so-called corecursive types (see Exercise 4).

We postpone discussion on the typing of process expression, cells, and configurations until the next lecture when we consider analogues of the progress and preservation theorems.

5 Example: A Pipeline

As a simple example for concurrency in this language we consider setting up a (very small) pipeline. We consider a sequence of bits

$$\mathit{bits} \cong (\mathtt{b0} : \mathit{bits}) + (\mathtt{b1} : \mathit{bits}) + (e : 1)$$

(which also happen to be isomorphic to binary numbers). Here is a function *flip* : *bits* $\to$ *bit* that just flips every bit. We use our pattern matching syntax with constructors

$$\begin{array}{lcl} B0 & : & \mathit{bits} \to \mathit{bits} \\ B1 & : & \mathit{bits} \to \mathit{bits} \\ E & : & \mathit{bits} \end{array}$$

$\mathit{flip} : \mathit{bits} \to \mathit{bits}$
$\mathit{flip}\ (B0\ y) = B1\ (\mathit{flip}\ y)$
$\mathit{flip}\ (B1\ y) = B0\ (\mathit{flip}\ y)$
$\mathit{flip}\ E = E$

In preparation for translation to process form, we write it out in more explicit notation. For the sake of brevity, we skip the fold/unfold constructors, which can easily be added.

$$\begin{array}{l} [\![\mathit{flip}]\!]\ d = \mathsf{case}\ d^W\ (\langle x, z\rangle \Rightarrow \mathsf{case}\ x^R\ (\ \mathtt{B0}\cdot y \Rightarrow d_1 \leftarrow (d_2 \leftarrow [\![\mathit{flip}]\!]\ d_2\ ; \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad d_3 \leftarrow (d_3^W \leftarrow y^R)\ ; \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad d_2^R.\langle d_3, d_1\rangle) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad z^W.\mathtt{B1}(d_1) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathtt{B1}\cdot y \Rightarrow \ldots \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathtt{E}\cdot y \Rightarrow \mathtt{E}\cdot y\ )\ ) \end{array}$$

Rather than addressing how fixed points are translated in their full generality, we merely derived a recursive specification for the process $[\![\mathit{flip}]\!]$.

We didn't do this in lecture, but we can apply some optimizations to see more easily how this process executes. First, since the definition of *flip* is closed, we assume its definition is stored in a fixed cell f_0 (never mind that the store is then circular). Also, we can replace the left-hand side below with the right-hand side

$$d_3 \leftarrow (d_3^W \leftarrow y^R)\ ; P \equiv [y/d_3]P$$

to arrive at

$$\begin{array}{l} !\mathsf{cell}\ f_0\ (\langle x, z\rangle \Rightarrow \mathsf{case}\ x^R\ \ (\ \mathtt{B0}\cdot y \Rightarrow d_1 \leftarrow f_0^R.\langle y^R, d_1\rangle\ ; \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad z^W.\mathtt{B1}(d_1) \\ \qquad\qquad\qquad\qquad\quad \mid \mathtt{B1}\cdot y \Rightarrow d_1 \leftarrow f_0^R.\langle y^R, d_1\rangle\ ; \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad z^W.\mathtt{B0}(d_1) \\ \qquad\qquad\qquad\qquad\quad \mid \mathtt{E}\cdot y \Rightarrow d_1 \leftarrow (d_1 \leftarrow y)\ ; \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad z^W.\mathtt{E}(y)\ ) \end{array}$$

Now imagine we would like to compute *flip* $(B0(B1(E)))$ The argument will be represent in memory by (again ignoring folds)

!cell c_2 $\langle\,\rangle$
!cell c_1 $(\texttt{B1}(c_2))$
!cell c_0 $(\texttt{B0}(c_2))$

The call to *flip* with destination d_0 would be

proc d_0 $(f_0^R.\langle c_0, d_0\rangle)$, cell d_0 _

When this process reads the contents c_0 and takes the `B0` branch, it allocates a new destination d_1 for the result of the recursive call, but meanwhile it can already write $\texttt{B0}(d_1)$ into the destination d_0. So we have

!cell f_0 $(\ldots)$

!cell c_2 $\langle\,\rangle$
!cell c_1 $(\texttt{B1}(c_2))$
!cell c_0 $(\texttt{B0}(c_2))$

!cell d_0 $(\texttt{B1}(d_1))$
cell d_1 _

proc d_1 $(f_0^R.\langle c_1, d_1\rangle)$

The significance in this state is that the process has already written part of the output (into destination d_0) *before* having read all of the input. For example, if this where the inner flip of of *flip* $(\mathit{flip}\,(B0(B1(E))))$ then the outer flip could now read the destination d_0 and output $\texttt{B1}(a_1)$ while the inner flip had only read one bit of input.

So we see that under a concurrent interpretation the composition *flip* ∘ *flip* behaves like a pipeline with two processes, *flip* ∘ *flip* ∘ *flip* behaves like a pipeline with three processes, etc.

Under a sequential interpretation, where $x \leftarrow P \; ; Q$ waits until P has written to destination x before Q starts executing, all recursive calls in *flip* would have to be finished before the first bit of output is written. When we compose two, the inner one has to finish entirely, writing out the whole sequence of bits before the outer one can start.

Exercises

Exercise 1 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

$$\mathit{stream}\ \alpha \quad\cong\quad (\texttt{hd} : \alpha) \mathbin{\&} (\texttt{tl} : \mathit{stream}\ \alpha)$$

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*):

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

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

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

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.

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

Write functions satisfying the following specifications:

1. *up_from* : $\mathit{nat} \to \mathit{stream}\ \mathit{nat}$ where *up_from* n generates the stream $n, n+1, n+2, \ldots$.
2. *alt* : $\forall \alpha.\ \mathit{stream}\ \alpha \to \mathit{stream}\ \alpha \to \mathit{stream}\ \alpha$ 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.

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

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

Exercise 2 For lazy records as introduced in Exercise 1 we introduce the following syntax in our language of expressions:

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

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.

Exercise 3 Explore what the rules and meaning of $\bot$ as the formal dual of 1 in the process language should be, including whichever of the following you find make sense. If something does not make sense somehow, please explain.

1. Write out the new forms of process expressions.

2. Provide the store-based dynamics for the new process expressions.
3. Show the typing rules for the new process expressions.
4. Reverse-engineer new functional expressions in our original language so they translate to your new process expression. Show the rules for typing and stepping the new constructs.
5. Summarize and discuss what you found.

Exercise 4 In our expression language the fold e constructor for elements of recursive type is eager. Explore a new *lazy* ravel e constructor, providing:

1. Typing rules for knit and a corresponding destructor (presumably an unravel or a case construct).
2. Stepping rules for the new forms of expressions.
3. A translation from the new forms of expressions to processes, extending the language of processes as needed
4. Transition rules for the new forms of processes.