Assignment 7
Concurrency and Laziness

15-814: Types and Programming Languages
Frank Pfenning

Due Tuesday, Dec 1, 2020

- `hw07.pdf` with your written solutions to the questions. You may exclude those questions whose answer is just code, but please leave a pointer to the code.
- `hw07.cbv` with the code, where the solutions to the questions are clearly marked and auxiliary code (either from lecture or your own) is included so it passes the LAMBDA implementation.

1 Shared Memory Concurrency

Task 1 (L20.1, 20 points) For lazy records (as a generalization of lazy pairs) 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.

Task 2 (L20.3, 20 points) In our expression language the $\mathsf{fold}\ e$ constructor for elements of recursive type is eager. Explore a new *lazy* $\mathsf{ravel}\ e$ constructor which has type $\delta\alpha.\,\tau$, providing:

1. Typing rules for ravel and a corresponding destructor (presumably an $\mathsf{unravel}$ or 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. Your translation should expose and exploit the duality between lazy and eager recursive types if such a duality exists.
4. Typing rules for the new forms of processes.
5. Transition rules for the new forms of processes.

Task 3 (L21.3, 20 points) Consider the type of tree where the information is kept only in the leaves:

$$\mathit{shrub}\ \alpha = \rho t.\, (\mathbf{branch} : t \times t) + (\mathbf{bud} : \alpha)$$

(i) Write a version of *mapreduce* that operates on shrubs and exhibits analogous concurrent behavior. You may use similar shortcuts to the ones we used in our implementation.

(ii) Write processes *forth* and *back* to translate between trees and shrubs while preserving the elements. Do they form an isomorphism? If not, do you see a simple modification to restore an isomorphism?

Task 4 (L21.4, 20 points) The sequential execution in Section L21.5 is *eager* in the sense that in $x \Leftarrow P\ ;\ Q$, P completes by writing to x before Q starts.

A lazy version, $x \leftharpoondown P\ ;\ Q$ would immediately start Q and suspend P until Q (or some process spawned by it) would try to read from x. We would still like it to be sequential in the sense that at most one process can take a step at any time.

Devise a semantics for $x \leftharpoondown P\ ;\ Q$ that exhibits the desired lazy behavior while remaining sequential. You may introduce new semantic objects or apply some transformation to P and/or Q, but you should strive for the simplest, most elegant solution to keep the dynamics simple.

2 Lazy Functional Programming

Task 5 (L22.2, 20 points) Write functions on streams as in Section L22.2 satisfying the specifications below.

(i) $\mathit{alt} : \forall \alpha.\, \mathit{stream}\ \alpha \to \mathit{stream}\ \alpha \to \mathit{stream}\ \alpha$ which alternates the elements from the two streams, starting with the first element of the first stream.

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

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

(iv) $\mathit{diag} : \forall \alpha.\, \mathit{stream}\ (\mathit{stream}\ \alpha) \to \mathit{stream}\ \alpha$ 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 and write auxiliary functions as needed.

In the LAMBDA implementation, you may choose the special case that $\alpha = \beta = \mathit{nat}$. In the absence of type constructors in LAMBDA, define types

$\mathit{stream} = \rho s.\, (\mathbf{hd} : \mathit{nat}) \mathbin{\&} (\mathbf{tl} : s)$
$\mathit{sstream} = \rho \mathit{ss}.\, (\mathbf{hd} : \mathit{stream}) \mathbin{\&} (\mathbf{tl} : \mathit{ss})$

where the first was already present in Section L22.2 and the second is needed for part (iv).

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, among the three definitions below of a stream transducer that adds 1 to every element, only the first definition would be lazy enough. The second definition (*succs*′) would be still terminating, but slighty too eager (for example, we may never access the element at the head of the resulting stream which would have been computed unnecessarily), while the third (succs″) would not even be terminating any more.

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

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

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

Here we have used $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ as syntactic sugar for $(\lambda x.\, e)\, e'$.

Task 6 (L22.3, 20 points) Following the style of object-oriented programming in Section L22.3 consider the types of queue

$$\begin{aligned}\mathit{queue}\ \alpha = \rho s.\quad & (\mathbf{enq} : \alpha \to \mathit{queue}\ \alpha) \\ \mathbin{\&}\ & (\mathbf{deq} : \quad (\mathbf{none} : \mathit{queue}\ \alpha) \\ & \qquad + (\mathbf{some} : \alpha \times \mathit{queue}\ \alpha))\end{aligned}$$

(i) Write a function

$$\mathit{reverse} : \forall\alpha.\, \mathit{stack}\ \alpha \to \mathit{stack}\ \alpha$$

that reverses the elements of the given stack.

(ii) Provide an implementation of queues

$$\mathit{queue\_stacks} : \forall\alpha.\, \mathit{stack}\ \alpha \to \mathit{stack}\ \alpha \to \mathit{queue}\ \alpha$$

where a queue is represented by a pair of stacks (see below).

(iii) Provide an empty queue

$$\mathit{queue\_new} : \forall\alpha.\, \mathit{queue}\ \alpha$$

One of your stacks should be the *input stack*. Elements to be enqueued should be pushed on this input stack. The second stack should be the *output stack*. Elements to be dequeued should be taken from the output stack. If the output stack happens to be empty but some elements remain on the input stack, reverse the input stack to become the new output stack. This technique is sometimes called *functional queues*.

As in Section L22.3, in the absence of type constructors in LAMBDA you may specialize the types of stacks and queues to $\alpha = \mathit{nat}$.