Assignment 3:
The Meaning of Laziness

15-312: Foundations of Programming Languages
Joshua Dunfield (joshuad@cs.cmu.edu)

Out: Thursday, September 19, 2002
Due: Thursday, September 26, 2002 (1:30 pm)

50 points total

In this assignment, we'll alter MinML to use call-by-name evaluation instead of call-by-value, and add lazy pairs and lazy lists (streams).

Call-by-name MinML + lazy pairs + streams

We start by adding several new kinds of syntax to the language of Assignment 2:

Construct	Concrete Syntax	Abstract Syntax
Lazy pairs	(e_1, e_2)	$\texttt{pair}(e_1, e_2)$
Left projection	$\texttt{fst}\ e$	$\texttt{fst}(e)$
Right projection	$\texttt{snd}\ e$	$\texttt{snd}(e)$
Empty stream	$\texttt{nil}$	$\texttt{nil}$
Cons	$e_1 : e_2$	$\texttt{cons}(e_1, e_2)$
Case on streams	$\texttt{case}\ e\ \texttt{of nil =>}\ e_1\ \texttt{|}\ x : s\ \texttt{=>}\ e_2$	$\texttt{case}(e, e_1, x.s.e_2)$

We'll stick with abstract syntax for the rest of the assignment.

The intended lazy interpretation of the new constructs is captured in the following extension of the notion of value.

$$\frac{}{\texttt{pair}(e_1, e_2)\ \textsf{value}}$$

$$\frac{}{\texttt{nil}\ \textsf{value}}$$

$$\frac{}{\texttt{cons}(e_1, e_2)\ \textsf{value}}$$

We must also add some typing rules. The complete set of typing rules is given in Figure 1. For simplicity the type-checking rules allow only streams of integers.

Here are two example programs. The first returns an infinite stream `0:1:2:3:...`:

```
(fun from (x: int): stream is
  x:(from (x+1))
end) 0
```

The second program defines a function that takes a stream and returns its length if it is finite, and otherwise fails to terminate:

```
let length = fun length (s: stream): int is
                 case s of
                  nil => 0
                | x:s' => 1 + length s'
             end
in
   length (1:2:3:nil)
end
```

Extending the dynamic semantics is your job. You must:

1. Rewrite the transition rules so that function application is done "by name", that is, the argument to a function is *not* reduced to a value before substitution occurs.

2. Add transition rules for each new construct. Pairs and streams should be lazy, as the definition of the value judgment at the top of the page indicates.

$$\frac{}{\Gamma_1, x{:}\tau, \Gamma_2 \vdash x : \tau}\ \textit{VarTyp}$$

$$\frac{}{\Gamma \vdash \texttt{num}(n) : \texttt{int}}\ \textit{NumTyp}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \textit{TrueTyp} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ \textit{FalseTyp}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}(e, e_1, e_2) : \tau}\ \textit{IfTyp}$$

$$\frac{\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}(\tau_1, \tau_2, f.x.e) : \tau_1 \to \tau_2}\ \textit{FunTyp}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{apply}(e_1, e_2) : \tau}\ \textit{AppTyp}$$

$$\frac{\Gamma \vdash e_1 : \tau_{o1} \quad \ldots \quad \Gamma \vdash e_n : \tau_{on}}{\Gamma \vdash o(e_1, \ldots, e_n) : \tau_o}\ \textit{OpTyp}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}(e_1, x.e_2) : \tau_2}\ \textit{LetTyp}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{pair}(e_1, e_2) : \tau_1 * \tau_2}\ \textit{PairTyp}$$

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \texttt{fst}(e) : \tau_1}\ \textit{FstTyp} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \texttt{snd}(e) : \tau_2}\ \textit{SndTyp}$$

$$\frac{}{\Gamma \vdash \texttt{nil} : \texttt{stream}}\ \textit{NilTyp} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{stream}}{\Gamma \vdash \texttt{cons}(e_1, e_2) : \texttt{stream}}\ \textit{ConsTyp}$$

$$\frac{\Gamma \vdash e : \texttt{stream} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, x : \texttt{int}, s : \texttt{stream} \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}(e, e_1, x.s.e_2) : \tau}\ \textit{CaseTyp}$$

Figure 1: Static semantics for MinML

Question 1 (15 points). Write down a dynamic semantics for MinML with call-by-name evaluation, lazy pairs, and streams (lazy lists). Make sure your semantics is deterministic: a given expression should either be a value, or it may step, but never both. Also, if an expression can make a step, this step should be uniquely determined. You are welcome to prove these properties, but don't turn in the proof.

Question 2 (35 points). Prove the Preservation and Progress theorems with respect to your semantics. Please state your induction hypothesis and possible lemmas carefully. Only show proof cases related to the `pair`, `snd`, `fst`, `nil`, `cons`, and `case` constructs. If you need them, you may assume *weakening* the *expression substitution* property for your system without proof.

(i) (Weakening) If $\Gamma_1, \Gamma_2 \vdash e' : \tau'$ the $\Gamma_1, x{:}\tau, \Gamma_2 \vdash e' : \tau'$.

(ii) (Expression Substitution)
If $\Gamma_1, x{:}\tau, \Gamma_2 \vdash e' : \tau'$ and $\cdot \vdash e : \tau$ then $\Gamma_1, \Gamma_2 \vdash \{e/x\}e' : \tau'$.