Assignment 3:
The Meaning of Laziness

15-312: Foundations of Programming Languages

Matt Moore (`mlmlm@cmu.edu`)

Out: Thursday, September 23, 2004
Due: Thursday, September 30, 2004 (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 (i.e. streams).

Call-by-name MinML + lazy pairs + streams

We start by adding two new types, a lazy pair type and a stream type, to the language defined in Assignment 2. Note that pairs as defined in this assignment differ from those defined in lecture. The concrete and abstract syntax for these new constructs is found below.

Construct	Concrete Syntax	Abstract Syntax
Type of lazy pairs	τ_1 `*` τ_2	`cross`(τ_1, τ_2)
Lazy pairs	`(`e_1`,` e_2`)`	`pair`(e_1, e_2)
Left projection	`fst` e	`fst`(e)
Right projection	`snd` e	`snd`(e)
Type of streams	τ `stream`	`stream`(τ)
Empty stream	`nil` : τ `stream`	`nil`(τ)
Cons	e_1`::`e_2	`cons`(e_1, e_2)
Case on streams	`case` e `of` `nil` `=>` e_1 `|` x`::`s `=>` e_2	`case`$(e, e_1, x.s.e_2)$

We will use abstract syntax for expressions in rest of the assignment, but continue to use concrete syntax for types (for the sake of readability).

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

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

$$\frac{}{\texttt{nil}(\tau)\ \mathsf{value}}$$

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

We must also add some typing rules. The complete set of typing rules is given in Figure **??**.

Consider the following two example programs. The first returns an infinite stream `0::1::2::3::`...

```
(rec from : int -> int stream => fn x : int =>
  x::(from (x+1))
) 0
```

The second program defines a function that takes a stream and returns its length if it is finite; otherwise, it fails to terminate. (Remember that `fun` is syntactic sugar for `rec` and `fn`.)

```
let length = fun length (s : int stream) : int =>
                 (case s of
                     nil => 0
                   | x::s' => 1 + length s')
in
   length (1::2::3::nil : int stream)
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. Let bindings should also be lazy.
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.

$$\frac{}{\Gamma_1, x{:}\tau, \Gamma_2 \vdash x : \tau}\ \textit{VarTyp} \qquad \frac{}{\Gamma \vdash \texttt{num}(n) : \texttt{int}}\ \textit{NumTyp}$$

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

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

$$\frac{\Gamma, x{:}\tau \vdash e : \tau}{\Gamma \vdash \texttt{rec}(\tau, x.e) : \tau}\ \textit{RecTyp}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fn}(\tau_1, x.e) : \tau_1 \to \tau_2}\ \textit{FnTyp}$$

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

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

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

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

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

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

$$\frac{\Gamma \vdash e : \tau_1\,\texttt{stream} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_1, s : \tau_1\,\texttt{stream} \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}(e, e_1, x.s.e_2) : \tau}\ \textit{CaseTyp}$$

Figure 1: Static semantics for MinML

Question 1 (10 points). Write down a dynamic semantics for MinML with call-by-name evaluation, lazy pairs, and streams (lazy lists). For those rules which remain unchanged from Assignment 2, you may include them by reference. 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 (30 points). Prove the Preservation and Progress theorems with respect to your semantics. Please carefully state your induction hypothesis and any lemmas that you use. Show only those proof cases in your theorems and lemmas related to the `pair`, `snd`, `fst`, `nil`, `cons`, and `case` constructs. If you need them, you may assume the *weakening* and *expression substitution* properties 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 $\Gamma_1, \Gamma_2 \vdash \{e/x\}e' : \tau'$.

An Alternate Formulation

We have made the type of the elements in a stream explicit in the type of the stream. Consider the following alternative typing rules for `cons` and `nil` where that is not the case.

$$\frac{}{\Gamma \vdash \texttt{nil} : \texttt{stream}}\ \textit{NilTyp}'$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{stream}}{\Gamma \vdash \texttt{cons}(e_1, e_2) : \texttt{stream}}\ \textit{ConsTyp}'$$

Question 3 (10 points). Give counterexamples to preservation and/or progress. Briefly identify where and how your safety proof from Question 2 would break down if you were to use these alternate rules.

Question 4 (EXTRA CREDIT, 20 points). Give counterexamples to equivalence between call-by-value and call-by-name semantics for function application in the following sense:

- Give a program where one semantics diverges while the other terminates.
- Give a program that terminates under both call-by-value and call-by-name but returns different results (modulo α-conversion, as usual).

Discuss in what sense call-by-value and call-by-name might nonetheless be considered equivalent.