15-851 COMPUTATION AND DEDUCTION

MODEL SOLUTION OF ASSIGNMENT 1
BRIGITTE PIENTKA
January 31, 2001

Exercise 2.1: Write Mini-ML programs for multiplication, exponentiation, subtraction, and a function that returns a pair of (integer) quotient and remainder of two natural numbers.

Solution:

$$
\begin{array}{lcl}
add & = & \mathbf{fix}\ f.\,\mathbf{lam}\ x.\,\mathbf{lam}\ y.\,\mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow y \,|\, \mathbf{s}\ x' \Rightarrow \mathbf{s}\ (f\,x'\,y) \\
sub & = & \mathbf{fix}\ f.\,\mathbf{lam}\ x.\,\mathbf{lam}\ y. \\
 & & \quad\quad \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \\
 & & \quad\quad\quad\quad |\, \mathbf{s}\ x' \Rightarrow \mathbf{case}\ y\ \mathbf{of}\ \mathbf{z} \Rightarrow x \,|\, \mathbf{s}\ y' \Rightarrow f\,x'\,y'. \\
mult & = & \mathbf{fix}\ f.\,\mathbf{lam}\ x.\,\mathbf{lam}\ y. \\
 & & \quad\quad \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \,|\, \mathbf{s}\ x' \Rightarrow add\,(f\,x'\,y\,)\,y \\
expo & = & \mathbf{fix}\ f.\,\mathbf{lam}\ x.\,\mathbf{lam}\ n. \\
 & & \quad\quad \mathbf{case}\ n\ \mathbf{of}\ \mathbf{z} \Rightarrow (\mathbf{s}\ \mathbf{z}) \,|\, \mathbf{s}\ n' \Rightarrow mult\,x\,(f\,x\,n') \\
 & & \\
quot & = & \mathbf{fix}\ f.\,\mathbf{lam}\ x.\,\mathbf{lam}\ y. \\
 & & \quad\quad \mathbf{case}\ sub\,x\,y\ \mathbf{of} \\
 & & \quad\quad\quad \mathbf{z} \Rightarrow \mathbf{case}\ sub\,y\,x\ \mathbf{of}\ \mathbf{z} \Rightarrow \langle \mathbf{s}\ \mathbf{z}, \mathbf{z}\rangle \,|\, \mathbf{s}\ x' \Rightarrow \langle \mathbf{z}, x\rangle \\
 & & \quad\quad |\, \mathbf{s}\ w \Rightarrow \mathbf{let}\ \mathbf{val}\ v = f\ (\mathbf{s}\ w)\ y\ \mathbf{in}\ \langle \mathbf{s}\ (\mathbf{fst}\ v), \mathbf{snd}\ v\rangle
\end{array}
$$

Exercise 2.13: Specify a call-by-name operational semantics for our language where the constructors are *lazy* that is they should not evaluate their arguments.

Solution: We start by defining lazy values. If we discover an expression $\mathbf{s}\ e$ then we reached a value as we will only evaluate e when needed. Similarly, a pair $\langle e_1, e_2\rangle$ is a value.

$$\frac{}{\mathbf{z}\ Lazy\_Val}\ \textsf{lval\_z} \qquad \frac{}{\mathbf{s}\ e\ Lazy\_Val}\ \textsf{lval\_s}$$

$$\frac{}{\mathbf{lam}\ x.e\ Lazy\_Val}\ \textsf{lval\_lam} \qquad \frac{}{\langle e_1, e_2\rangle\ Lazy\_Val}\ \textsf{lval\_pair}$$

We proceed by revising the operational semantics of Mini-ML.

$$\frac{}{\mathbf{z} \stackrel{l}{\hookrightarrow} \mathbf{z}}\ \mathsf{evl\_z} \qquad \frac{}{\mathbf{s}\ e \stackrel{l}{\hookrightarrow} \mathbf{s}\ e}\ \mathsf{evl\_s}$$

$$\frac{e \stackrel{l}{\hookrightarrow} \mathbf{z} \qquad e_1 \stackrel{l}{\hookrightarrow} v}{\mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \,|\, \mathbf{s}\ x' \Rightarrow e_2 \stackrel{l}{\hookrightarrow} v}\ \mathsf{evl\_case\_z} \qquad \frac{e \stackrel{l}{\hookrightarrow} \mathbf{s}\ e' \qquad [e'/x']e_2 \stackrel{l}{\hookrightarrow} v}{\mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \,|\, \mathbf{s}\ x' \Rightarrow e_2 \stackrel{l}{\hookrightarrow} v}\ \mathsf{evl\_case\_s}$$

$$\frac{}{\mathbf{lam}\ x.e \stackrel{l}{\hookrightarrow} \mathbf{lam}\ x.e}\ \mathsf{evl\_lam} \qquad \frac{e_1 \stackrel{l}{\hookrightarrow} \mathbf{lam}\ x.e' \qquad [e_2/x]e' \stackrel{l}{\hookrightarrow} v}{e_1\ e_2 \stackrel{l}{\hookrightarrow} v}\ \mathsf{evl\_app}$$

$$\frac{}{\langle e_1, e_2 \rangle \stackrel{l}{\hookrightarrow} \langle e_1, e_2 \rangle}\ \mathsf{evl\_pair}$$

$$\frac{e \stackrel{l}{\hookrightarrow} \langle e_1, e_2 \rangle \qquad e_1 \stackrel{l}{\hookrightarrow} v}{\mathbf{fst}\ e \stackrel{l}{\hookrightarrow} v}\ \mathsf{evl\_fst} \qquad \frac{e \stackrel{l}{\hookrightarrow} \langle e_1, e_2 \rangle \qquad e_2 \stackrel{l}{\hookrightarrow} v}{\mathbf{snd}\ e \stackrel{l}{\hookrightarrow} v}\ \mathsf{evl\_snd}$$

The evl_letn rule does not change as it already is lazy, i.e. it does not evaluate the argument x. In order to force the evaluation of an expression, we choose to include the evl_letv rule.

$$\frac{e_1 \stackrel{l}{\hookrightarrow} v_1 \qquad [v_1/x]e_2 \stackrel{l}{\hookrightarrow} v}{\mathbf{let\ val}\ x = e_1\ \mathbf{in}\ e_2 \stackrel{l}{\hookrightarrow} v}\ \mathsf{evl\_letv} \qquad \frac{[e_1/u]e_2 \stackrel{l}{\hookrightarrow} v}{\mathbf{let\ name}\ u = e_1\ \mathbf{in}\ e_2 \stackrel{l}{\hookrightarrow} v}\ \mathsf{evl\_letn}$$

The evl_fix rule stays the same.

$$\frac{[\mathbf{fix}\ e/x]\ e \stackrel{l}{\hookrightarrow} v}{\mathbf{fix}\ e \stackrel{l}{\hookrightarrow} v}\ \mathsf{evl\_fix}$$

Theorem 1 (Value Soundness)**.** *If* $\mathcal{D} :: e \stackrel{l}{\hookrightarrow} v$ *then* $\mathcal{E} :: v\ Lazy\_Val$.

Proof. The proof follows by induction over the structure of the deduction $\mathcal{D} :: e \stackrel{l}{\hookrightarrow} v$. We will only show a few typical cases.

Case: $\mathcal{D} = \dfrac{}{\mathbf{z} \stackrel{l}{\hookrightarrow} \mathbf{z}}\ \mathsf{evl\_z}$. Then $\mathbf{z}\ Lazy\_Val$ by the rule lval_z.

Case: $\mathcal{D} = \dfrac{}{\mathbf{s}\ e \stackrel{l}{\hookrightarrow} \mathbf{s}\ e}\ \mathsf{evl\_s}$. Then $\mathbf{s}\ e\ Lazy\_Val$ by the rule lval_s.

Case: $\mathcal{D} = \dfrac{}{\mathbf{lam}\ x.e \stackrel{l}{\hookrightarrow} \mathbf{lam}\ x.e}\ \mathsf{evl\_lam}$.

Then **lam** $x.e$ $Lazy\_Val$ by the rule lval_lam.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ e_1 \overset{l}{\hookrightarrow} \textbf{lam}\ x.e' & [e_2/x]e' \overset{l}{\hookrightarrow} v \end{array}}{e_1\ e_2 \overset{l}{\hookrightarrow} v}\ \textsf{evl\_app}$

The induction hypothesis on $\mathcal{D}_2$ yields a deduction $\mathcal{E} :: v\ Lazy\_Val$.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ e \overset{l}{\hookrightarrow} \langle e_1, e_2 \rangle & e_1 \hookrightarrow v \end{array}}{\textbf{fst}\ e \overset{l}{\hookrightarrow} v}\ \textsf{evl\_fst}$

The induction hypothesis on $\mathcal{D}_2$ yields a deduction $\mathcal{E} :: v\ Lazy\_Val$.

□

Exercise 2.14 - Part 1: Prove that $v\ Value$ is derivable if and only if $v \hookrightarrow v$ is derivable. That is, values are exactly those expressions that evaluate to themselves.

Solution: Theorem 2. *If $\mathcal{D} :: v\ Value$ then $\mathcal{E} :: v \hookrightarrow v$.*

Proof. By induction over the structure of the deduction $\mathcal{D} :: v\ Value$.

Case: $\mathcal{D} = \dfrac{}{\textbf{z}\ Value}\ \textsf{val\_z}$. Then $\textbf{z} \hookrightarrow \textbf{z}$ by the rule ev_z.

Case: $\mathcal{D} = \dfrac{\begin{array}{c} \mathcal{D}_1 \\ v\ Value \end{array}}{\textbf{s}\ v\ Value}\ \textsf{val\_s}$

The induction hypothesis on $\mathcal{D}_1$ yields a deduction $\mathcal{E}_1 :: v \hookrightarrow v$. Using the inference rule ev_s we conclude that $\textbf{s}\ v \hookrightarrow \textbf{s}\ v$.

Case: $\mathcal{D} = \dfrac{}{\textbf{lam}\ x.e\ Value}\ \textsf{val\_lam}$.

Then **lam** $x.e \hookrightarrow$ **lam** $x.e$ by the rule ev_lam.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ v_1\ Value & v_2\ Value \end{array}}{\langle v_1, v_2 \rangle\ Value}\ \textsf{val\_pair}$

$v_1 \hookrightarrow v_1$	by induction hypothesis on $\mathcal{D}_1$
$v_2 \hookrightarrow v_2$	by induction hypothesis on $\mathcal{D}_2$
$\langle v_1, v_2 \rangle \hookrightarrow \langle v_1, v_2 \rangle$	by rule ev_pair

□

Theorem 3. *If $\mathcal{E} :: v \hookrightarrow v$ then $\mathcal{D} :: v\ Value$.*

Proof. Follows immediately from the value-soundness theorem **Theorem 2.1** p 19 of the lecture notes. □

Exercise 2.14 - Part 2: Write a Mini-ML function *observe* : $\mathsf{nat} \rightarrow \mathsf{nat}$ that, given a lazy value of type nat, returns the corresponding eager value if it exists.

Solution:

There are two possible ways to observe the value of a lazy expression. The first solution uses the **let val** construct to force the evaluation of a lazy expression.

$$\mathit{observe} \quad = \quad \mathbf{fix}\ f.\mathbf{lam}\ x.\mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow \mathbf{z} \,|\, \mathbf{s}\ x' \Rightarrow \mathbf{let\ val}\ v = f\ x'\ \mathbf{in}\ \mathbf{s}\ v.$$

The second solution is based on continuations. The basic idea is the following: any function $f : t \rightarrow s$ can be rewritten into a function f' of type $t \rightarrow (s \rightarrow b) \rightarrow b$. In contrast to f, the function f' takes an extra function as an argument, called a *continuation*, which accumulates the results. To use the function f' to compute the original function f, we give it the *initial continuation* which is often the identity function as an argument. Applying this idea to define *observe* we first define a function *observe*$'$ which takes x and a continuation k as an argument. In the base case, we just call the continuation k applied to $\mathbf{z}$. In the recursive case, we apply the successor function to the result of the continuation. Note that the successor function will be only applied to values once it is executed.

$$\begin{aligned} \mathit{observe}' &= \mathbf{fix}\ f.\mathbf{lam}\ x.\mathbf{lam}\ k.\mathbf{case}\ x\ \mathbf{of}\ \mathbf{z} \Rightarrow k\ \mathbf{z} \,|\, \mathbf{s}\ x' \Rightarrow f\ x'\ (\mathbf{lam}\ v.k\ (\mathbf{s}\ v)). \\ \mathit{observe} &= \mathbf{lam}\ x.\ \mathit{observe}'\ x\ (\mathbf{lam}\ v.v). \end{aligned}$$

Let us consider the following evaluation: *observe*$'$ $\mathbf{s}\ (\mathbf{s}\ ((\lambda x.x)\mathbf{z}))\ k$.

first rec. call:	*observe*$'$	$(\mathbf{s}\ ((\lambda x.x)\ \mathbf{z}))$	$(\mathbf{lam}\ v_1.k\ (\mathbf{s}\ v_1))$
sec. rec. call :	*observe*$'$	$((\lambda x.x)\ \mathbf{z})$	$(\mathbf{lam}\ v_2.(\mathbf{lam}\ v_1.k\ (\mathbf{s}\ v_1))\ (\mathbf{s}\ v_2))$

Now *observe*$'$ will evaluate $((\lambda x.x)\ \mathbf{z})$ to $\mathbf{z}$ and reach the base case where we need to compute $(\mathbf{lam}\ v_2.(\mathbf{lam}\ v_1.k(\mathbf{s}\ v_1))(\mathbf{s}\ v_2))\ \mathbf{z}$.