

## Exercise Sheet 6

Attempt as many questions as you can during the exercise class, and work on the remainder at home. Hand in your solutions to the tutor in the Friday exercise class, on 19 Mar, 2010.

### Exercise 1: Call-by-name reduction

- a. The first line of call-by-name reduction in Handout 7, paragraph 3, reads:

$$\text{and } (\text{not } (\text{null } l)) ((\text{car } l) > 0) \longrightarrow_{\beta}^* \text{if } (\text{not } (\text{null } l)) ((\text{car } l) > 0) \text{ false}$$

Write down each of the  $\beta$ -reduction steps involved in this reduction. For each step, notate the redex being reduced in square brackets and verify that the reduction context is a valid call-by-name context (according to the grammar given in paragraph 7).

- b. In the same block, there is allusion to a reduction sequence with the following effect:

$$\text{not } (\text{null } l) \longrightarrow_{\beta, \delta}^* \text{true}$$

Expand this out into individual reduction steps and verify that the reduction contexts are valid call-by-name contexts.

Recall that the function *not* is defined as  $\lambda p. \text{if } p \text{ false true}$ .

### Exercise 2: Call-by-value reduction

The function *or* is defined as follows:

$$\text{or} = \lambda p. \lambda q. \text{if } p \text{ true } q$$

- a. Using the values  $x = 4$  and  $n = 2$ , evaluate the following term under call by value reduction:

$$\text{or } (n = 0) (x/n = 2)$$

At each step, annotate your choice of redex in square brackets and verify that the reduction context is a valid call-by-value context (as per the grammar in paragraph 9).

- b. Redo the evaluation of the above expression for the values  $x = 4$  and  $n = 0$ .

### Exercise 3: Reflection on Call-by-value

Consider the fact that all primitive functions are called by “value”, i.e., their arguments are evaluated ahead of time. In particular, the primitive function “if” acts in this way. Do you foresee any problems with this scheme? Write down a recursive function definition, such as that of factorial, and contemplate how its evaluation would proceed in this regime.

(PTO)

#### Exercise 4: Delayed evaluation

Scheme is a call-by-value programming language. However, it introduced two constructs called **delay** and **force** in order to facilitate the use of infinite data structures. The syntax is:

$$M ::= x \mid c \mid \lambda x. M' \mid M_1 M_2 \mid \mathbf{delay} M' \mid \mathbf{force} M'$$

The idea is that the evaluation of (**delay**  $M'$ ) does nothing. (It represents the “delaying” of the evaluation of  $M'$ .) When the **force** operation is applied to a **delay**-term, then the evaluation of the delayed term is carried out. The reduction rule for this is:

$$\mathbf{force} (\mathbf{delay} M) \longrightarrow M$$

Using these constructs, the function for an infinite list of integers can be defined as follows:

$$\mathit{ints} \ n = \mathit{cons} \ n \ (\mathbf{delay} \ (\mathit{ints} \ (n + 1)))$$

- Propose a definition for the call-by-value reduction contexts for this extended language.
- Verify that your design allows terms such as  $\mathit{car} \ (\mathbf{force} \ (\mathit{cdr} \ (\mathit{ints} \ 2)))$  to be reduced to value terms (with evaluation terminating finitely).
- Suppose we regard **delay** and **force** as syntactic sugar for basic lambda calculus terms as follows:

$$\begin{aligned} \mathbf{delay} \ M &= \lambda x. M \\ \mathbf{force} \ N &= N \ 0 \end{aligned}$$

Using the standard set-up for call-by-value, do we get the desired results for **delay** and **force**?