

## Class Test 1

All the questions in this test deal with the following mystery function *times* written in applied lambda calculus:

$$\text{times} = (\lambda n. \lambda f. \text{if } (n = 0) (\lambda x. x) \\ (\lambda x. \text{times } (n - 1) f (f x)))$$

### 1. Types

Calculate the principal type (most general type) of *times*.

[20%]

(At a minimum, you should guess the correct type of *times*.)

$$\mathbf{int} \rightarrow (t \rightarrow t) \rightarrow t \rightarrow t$$

To calculate the principal type of *times*, we start by assuming a type  $t_1$  for the parameter  $n$  and a type  $t_2$  for the parameter  $f$ . Since this is a recursive definition, we also assume a type  $t_0$  for the recursive occurrence of *times*. (But, after deriving a type for the whole lambda abstraction, we should place a constraint that  $t_0$  is equal to the whole type.)

Since  $n$  is used as an integer, we get  $t_1 = \mathbf{int}$ .

Assume that “if” is of type  $\mathbf{bool} \rightarrow t_3 \rightarrow t_3 \rightarrow t_3$  for a new type variable  $t_3$ . (This is a generic instance of the polymorphic type of “if”.) Since the then-branch  $\lambda x. x$  is of type  $t_4 \rightarrow t_4$ , we get that  $t_3 = t_4 \rightarrow t_4$ . The else-branch must also be of type  $t_4 \rightarrow t_4$ . We can use this fact to reduce our calculations. (The longer route would have been to derive a type for the else-branch independently.) Assume that  $x$  is of type  $t_4$ . The application  $(f x)$  gives the constraint  $t_2 = t_4 \rightarrow t_5$ , for a new type variable  $t_5$ , and the type of the application is  $t_5$ . So, the recursive call of *times* has the type  $t_0 = \mathbf{int} \rightarrow (t_4 \rightarrow t_5) \rightarrow t_5 \rightarrow t_4$ . [Why?  $\mathbf{int}$  is the type  $(n - 1)$ ;  $(t_4 \rightarrow t_5)$  is the type of  $f$ ,  $t_5$  is the type of  $(f x)$ , and the result type  $t_4$  is the result type expected for the else branch of type  $t_4 \rightarrow t_4$ .]

The type of the whole lambda abstraction defining *times* is  $\mathbf{int} \rightarrow (t_4 \rightarrow t_5) \rightarrow t_4 \rightarrow t_4$ .

This must be equal to the assumed type for the recursive call of *times*, viz.,  $\mathbf{int} \rightarrow (t_4 \rightarrow t_5) \rightarrow t_5 \rightarrow t_4$ . This gives the equation  $t_4 = t_5$ . So, the type of *times* is  $\mathbf{int} \rightarrow (t_5 \rightarrow t_5) \rightarrow t_5 \rightarrow t_5$ , and we can rename  $t_5$  to just  $t$  to obtain the type mentioned above.

**Remark:** If the *times* function is defined using the **Y** combinator as:

$$\text{times} = \mathbf{Y} (\lambda \text{times}. \lambda n. \lambda f. \text{if } (n = 0) (\lambda x. x) \\ (\lambda x. \text{times } (n - 1) f (f x)))$$

the result will be exactly the same. Since **Y** has the polymorphic type  $\forall t. (t \rightarrow t) \rightarrow t$ , we use a generic instance of it, say,  $(t_0 \rightarrow t_0) \rightarrow t_0$ . This means that the type for the recursive call *times* is  $t_0$  and the type of the lambda abstraction  $\lambda n. \lambda f. \dots$  is also  $t_0$ . So,  $t_0$  is found to be a type of the form  $\mathbf{int} \rightarrow (t_5 \rightarrow t_5) \rightarrow t_5 \rightarrow t_5$ .

### 2. Lambda calculus comprehension

Write a one-sentence description of what the *times* function is, i.e., what does it return when applied to arguments  $n$  and  $f$  in general?

You might want to analyse the function carefully and work through some examples of its use in order to figure this out. (It is not expected to be obvious.)

[20%]

If  $n \geq 0$ , it returns the function  $f^n$ , i.e.,  $f \circ \dots \circ f$  iterated  $n$  times.

If  $n < 0$ , it doesn't terminate.

### 3. Reduction semantics

Reduce the following term to normal form:

**let** *inc* =  $\lambda z. z + 1$   
**in** *times 3 inc x*

Assume that the **Y** combinator can be reduced using the rule  $\mathbf{Y} M \rightarrow M (\mathbf{Y} M)$ .

You can use any reduction strategy. You need not show all the steps of the reduction in full glory, but include enough detail to make sure that your reductions are correct. Abbreviate terms as necessary to keep your work compact and manageable. [20%]

Eliding the enclosing **let** form:

	<i>times 3 inc x</i>
$\rightarrow_{\beta, \delta}^*$	$(\lambda x. \textit{times 2 inc (inc x)}) x$
$\rightarrow_{\beta}$	<i>times 2 inc (inc x)</i>
$\rightarrow_{\beta, \delta}^*$	$(\lambda x. \textit{times 1 inc (inc x)}) (\textit{inc x})$
$\rightarrow_{\beta}$	<i>times 1 inc (inc (inc x))</i>
$\rightarrow_{\beta, \delta}^*$	$(\lambda x. \textit{times 0 inc (inc x)}) (\textit{inc (inc x)})$
$\rightarrow_{\beta}$	<i>times 0 inc (inc (inc (inc x)))</i>
$\rightarrow_{\beta, \delta}^*$	$(\lambda x. x) (\textit{inc (inc (inc x))})$
$\rightarrow_{\beta}$	<i>inc (inc (inc x))</i>
$\rightarrow_{\beta}$	<i>inc (inc (x + 1))</i>
$\rightarrow_{\beta}$	<i>inc ((x + 1) + 1)</i>
$\rightarrow_{\beta}$	$((x + 1) + 1) + 1$

### 4. Lambda calculus comprehension

Given below is a function *m* defined using *times*:

$m = \lambda x. \lambda y. \mathbf{let} a = \lambda z. z + y$   
**in** *times x a 0*

What is the value of *m 3 4*?

[10%]

The value is 12.

*m 3 4* reduces to **let** *a* =  $\lambda z. z + 4$  **in** *times 3 a 0*. The calculation of this expression is similar to that of question 3. It reduces to  $((0 + 4) + 4) + 4$ .

Give a one-sentence description of the function. *m*

[10%]

*m x y* multiplies *x* by *y*. (It does this by adding *y* to zero *x* times.)

### 5. Lambda calculus

Define a function *drop*, which takes as arguments an integer *n* and a list *l*, and returns the list obtained by dropping the first *n* elements of *l*. For example *drop 2 (list 12345)* should have the value **(list 345)**.

You must define *drop* using the function *times*, without using recursion directly.

[20%]

*drop n l* = *times n cdr l*

or equivalently:

*drop n* = *times n cdr*