Resource Analysis: Problem Set 2

Jan Hoffmann
Carnegie Mellon University

January 27, 2016

Due before 1:30pm on Monday, February 1

2.1 (16 Points) Analysis of Insertion Sort

Insertion sort for lists of lists can be implemented in OCaml as follows.

```
let rec compare_list l1 l2 =
  match l1 with
  | [] → true
  | x::xs →
     match l2 with
     | [] → false
     | y::ys →
        if x = y then
          compare_list xs ys
        else
          x < y

let rec insert le x l =
  match l with
  | [] → [x]
  | y::ys →
     if le y x then y::insert le x ys
     else x::y::ys

let rec isort le l =
  match l with
  | [] → []
  | x::xs → insert le x (isort le xs)

let isort_list = isort compare_list
```

The function *isort_list* sorts lists in lexicographical order. Use recurrence relations to derive an asymptotic worst-case bound for *isort_list* in the same way we derived a bound for quick sort in the lecture.

a) Define a system of recurrence relation so that there is one recurrence relation for *isort_list* and for every function that can called by *isort_list*.

b) Solve the system of recurrence relations in a bottom-up manner.

Remarks:

Cost model: Derive a bound on the *number of function calls* that are performed by a funcstions. Constructor calls such as :: do not count as function calls.

Higher-order functions: It is up to you how to treat higher-order functions. One possibility is to use higher-order recurrence relations like $T(n, f) = f(n)$. Another possibility is to simply model first-order functions such as *insert compare_list* only.

Polymorphism: In OCaml, *isort_list* has the type *'a list list → 'a list list*. Assume for the analysis the type *int list list → int list list*. In particular, you can account constant time for the operations < and = in the function *insert*.

2.2 (8 Points) Amortized Analysis I (Potential Method)

Consider a sequence of operations $f_1, \ldots, f_n$ in which the cost of the ith operation is defined by

$$cost(f_i) = \begin{cases} i & \text{if } i = 2^k \text{ for some } k \\ 1 & \text{otherwise} \end{cases}$$

Use the potential method to determine the amortized cost per operation.

2.3 (12 Points) Amortized Analysis II (Functional Queue)

A simple and elegant way of implementing a queue in a functional programming language is to use two lists.

```
type 'a queue = Queue of 'a list * 'a list

let empty = Queue ([], [])

let enqueue (Queue (inq, outq)) x =
     Queue (x::inq, outq)

let rec rev_append l1 l2 =
  match l1 with
  | [] → l2
  | x::xs → rev_append xs (x::l2)

let rev l = rev_append l []

let rec dequeue (Queue (inq, outq)) =
  match outq,inq with
  | [], []  → raise Queue_empty
  | [], _   → dequeue (Queue ([],rev inq))
  | y::ys,_ → (Queue (inq, ys), y)
```

a) Briefly describe the idea of the *queue* data structure.

b) How many *cons* operations (uses of *::*) are performed by *dequeue* in the worst-cose?

c) Consider a sequence of n *enqueue* or *dequeue* operations of the form

```
let b1 in
let b2 in
...
let bk in
()
```

where

$$b_i \equiv \quad q = enqueue\ q\ n \quad \text{for an integer } n \quad \text{or} \quad b_i \equiv \quad (q,\_) = dequeue\ q$$

Use the potential method to determine the amortized number of *cons* operations that is performed per operation. (Clearly define the potential function that you are using.)

d) Consider the following OCaml expression.

```
let q = Queue ([3;2;1],[]) in
let (_,_) = dequeue q in
let (_,_) = dequeue q
```

How many *cons* operation are performed if the above code is evaluated? What program property is necessary to make the analysis you performed in Part (c) applicable?