Assignment 3:
(Implicit) Computation Complexity

15-819: Foundations of Quantitative Program Analysis (Fall 2019)

Out: Friday, October 8, 2019
Due: Friday, October 22, 2019 11:59pm EDT

1 System BC with Lists

Recall System BC with binary numerals from lecture. In this problem, you will extend System BC to lists and implement a sorting algorithm. Recall the type rule for recursive iterations on numerals.

$$\frac{\Delta;\cdot \vdash e : \mathrm{nat} \qquad \Delta;\Gamma \vdash e_0 : \mathrm{nat} \qquad \Delta, x_1 : \mathrm{nat};\Gamma, y_1 : \mathrm{nat} \vdash e_1 : \mathrm{nat} \qquad \Delta, x_2 : \mathrm{nat};\Gamma, y_2 : \mathrm{nat} \vdash e_2 : \mathrm{nat}}{\Delta;\Gamma \vdash \mathrm{rec}\{e_0; x_1, y_1.e_1; x_2, y_2.e_2\}(e) : \mathrm{nat}} \;(\text{BC:R}\textsc{ec})$$

We discussed in lecture that a generalization of the rule to recursion at higher types would lead to a language in which we can implement functions with super-polynomial complexity.

So assume that we keep all other type rules but replace the rule BC:REC with the rule BC:REC2 below. We call the resulting programming language System BC2.

$$\frac{\Delta;\cdot \vdash e : \tau \qquad \Delta;\Gamma \vdash e_0 : \mathrm{nat} \qquad \Delta, x_1 : \mathrm{nat};\Gamma, y_1 : \tau \vdash e_1 : \tau \qquad \Delta, x_2 : \mathrm{nat};\Gamma, y_2 : \tau \vdash e_2 : \tau}{\Delta;\Gamma \vdash \mathrm{rec}\{e_0; x_1, y_1.e_1; x_2, y_2.e_2\}(e) : \tau} \;(\text{BC:R}\textsc{ec}2)$$

Task 1.1 (10 pts). Find a function $f : \mathbb{N}^k \to \mathbb{N}$ that cannot be computed in polynomial time and show that it is implementable in System BC2 (by providing an implementation).

Hint: A function that growth exponentially is not commutable in polynomial time.

So let us go back to System BC with the rule BC:REC. The first step for implementing our sorting algorithm is to implement a comparison function for binary numerals.

Task 1.2 (10 pts). Implement a comparison function $leq : \square\mathrm{nat} \to \square\mathrm{nat} \to \mathrm{nat}$ such that $leq(\widetilde{n})(\widetilde{m})$ returns z if $n > m$ and $\mathrm{s}_0(\mathrm{z})$.

Our next goal is to extend System BC to lists. So we define types and expressions as follows.

$$\begin{array}{lll} \tau & ::= & \mathrm{nat} \\ & & \tau_1 \to \tau_2 \\ & & \square\tau_1 \to \tau_2 \\ & & L(\tau) \end{array}$$

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & & \mathrm{nil} & \mathsf{nil} \\
 & & \mathrm{cons}(e_1;e_2) & \mathsf{cons}\,(e_1,e_2) \\
 & & \mathrm{case}_{\mathrm{L}}\{e_0;x_1,x_2.e_1\}(e) & \mathsf{case}\ e\,\{\mathsf{nil} \hookrightarrow e_0 \mid \mathsf{cons}\,(x_1,x_2) \hookrightarrow e_1\} \\
 & & \mathrm{rec}_{\mathrm{L}}\{e_0;x_1,x_2,y.e_1\}(e) & \mathsf{rec}\ e\,\{\mathsf{nil} \hookrightarrow e_0 \mid \mathsf{cons}\,(x_1,x_2)\ \mathsf{with}\ y \hookrightarrow e_1\}
\end{array}
$$

The case analysis for lists plays the same role as the conditional for binary numbers: It is can be applied to safe lists while the recurser can only be applied to modal lists. This expressivity seems to be required to implement a sorting algorithm.

The dynamic semantics of lists is defined by the following rules.

$$
\frac{}{\mathrm{nil} \Downarrow \mathrm{nil}}\ (\text{E:Nil})
\qquad
\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{\mathrm{cons}(e_1;e_2) \Downarrow \mathrm{cons}(v_1;v_2)}\ (\text{E:Cons})
$$

$$
\frac{e \Downarrow \mathrm{nil} \qquad e_0 \Downarrow v}{\mathrm{case}_{\mathrm{L}}\{e_0;x_1,x_2.e_1\}(e) \Downarrow v}\ (\text{E:MatL-N})
\qquad
\frac{e \Downarrow \mathrm{cons}(v_1;v_2) \qquad [v_1,v_2/x_1,x_2]e_1 \Downarrow v}{\mathrm{case}_{\mathrm{L}}\{e_0;x_1,x_2.e_1\}(e) \Downarrow v}\ (\text{E:MatL-C})
$$

$$
\frac{e \Downarrow \mathrm{nil} \qquad e_0 \Downarrow v}{\mathrm{rec}_{\mathrm{L}}\{e_0;x_1,x_2,y.e_1\}(e) \Downarrow v}\ (\text{E:RecL-N})
$$

$$
\frac{e \Downarrow \mathrm{cons}(v_1;v_2) \qquad \mathrm{rec}_{\mathrm{L}}\{e_0;x_1,x_2,y.e_1\}(v_2) \Downarrow v_r \qquad [v_1,v_2,v_r/x_1,x_2,y]e_1 \Downarrow v}{\mathrm{rec}_{\mathrm{L}}\{e_0;x_1,x_2,y.e_1\}(e) \Downarrow v}\ (\text{E:RecL-C})
$$

Task 1.3 (10 pts). Define the static semantics of lists (4 type rules) so that all functions you can define are implementable in polynomial time.

Hint: You can allow recursion at all "data types", that is, types that do not contain arrows.

Task 1.4 (10 pts). Implement a function $\mathsf{sort} : \Box L(\mathrm{nat}) \to L(\mathrm{nat})$ that sorts a list in ascending order. You can use your previously defined comparison function.[1]

What is the (asymptotic) complexity of your solution?

Hint: Remember that you can use the input multiple times. On way to implement the function is to ensure that the i-th recursion returns the list of the i largest numbers in ascending order.

2 Functional Queue Revisited

In this problem, we are repeating the analysis of the queue from Assignment 1. But this time we use type-based amortized resource analysis.

In the OCaml code below, we have inserted *tick* expressions to count the number of cons (*::*) operations. Moreover, the code is in share-let normal form.

First, consider the reverse function for lists *rev*.

[1] You can get extra credit if you can implement a sorting algorithm that doesn't use the syntactic form for case analysis on lists. However, I don't think that this is possible.

```
let rec rev_append (l1, l2) =
  match l1 with
  | [] → l2
  | x::xs →
     let _ = Raml.tick 1.0 in
     let l2' = x::l2 in
     rev_append (xs, l2')

let rev l =
  let nil = [] in
  rev_append (l, nil)
```

Task 2.1 (4 pts). Provide resource-annotated types for the functions *rev_append*: $L(\text{bool}) \times L(\text{bool}) \to L(\text{bool})$ and *rev*: $L(\text{bool}) \to L(\text{bool})$ that can be derived using the type system for linear amortized resource analysis from lecture.

Task 2.2 (10 pts). Using the type system for linear amortized resource analysis from lecture, give type derivations of the types you provided for *rev_append* and *rev* in the previous task.

Task 2.3 (6 pts). Give a concise description of the set of all annotated function types that are derivable for the function *rev_append*.

Now consider the following variations of the queue implementation.

```
let enqueue (inq, outq) x =
  let _ = Raml.tick 1.0 in
  let inq' = x::inq in
  (inq', outq)

let rec dequeue (inq, outq) =
  match outq with
  | [] →
     begin
       match inq with
       | []  →
          let no_elem = [] in
          let empty_queue =
            let nil1 = [] in
            let nil2 = [] in
            (nil1,nil2)
          in
          (empty_queue, no_elem)
       | x::xs →
          let nil = [] in
          let inq_rev = rev inq in
          dequeue (nil,inq_rev)
     end
  | y::ys →
     let queue = (inq, ys) in
```

```
let nil = [] in
let _ = Raml.tick 1.0 in
let elem = y::nil in
(queue,elem)
```

Task 2.4 (4 pts). Give derivable resource-annotated types for the functions *enqueue*, and *dequeue*. *You don't have to provide the type derivation.*

Task 2.5 (10 pts). Give resource-annotated type derivations for the functions *a* and *b* below. Insert sharing expressions if needed.

```
let a =
  let qu = ([],[]) in
  let qu = enqueue (qu, 1) in
  let qu = enqueue (qu, 2) in
  let qu = enqueue (qu, 3) in
  dequeue qu

let b =
  let qu = ([],[]) in
  let qu = enqueue (qu, 1) in
  let qu = enqueue (qu, 2) in
  let qu = enqueue (qu, 3) in
  let _ = dequeue qu in
  dequeue qu
```

Task 2.6 (6 pts). Give a derivable resource-annotated type for the function *enq_or_deq*. *You don't have to provide the type derivation.*

```
let rec enq_or_deq (l,queue) =
  match l with
  | [] → queue
  | x::xs →
     if x then
       let queue' = enqueue (queue,x) in
       enq_or_deq (xs, queue')
     else
       let (queue',_) = dequeue queue in
       enq_or_deq (xs, queue')
```