Citius, Altius, Fortius

Rohan Bhalla and S. Arun-Kumar

October 28, 2013

1. A **sublist** of a list L is any list obtained by deleting some elements from L. For example, [], [1], [2] and $[1, 2]$ are all sublists of $[1, 2]$. However $[2, 1]$ is not a sublist of $[1, 2]$. Design a *higher order function* $sublists : ''a\ list \rightarrow ''a\ list\ list$ that constructs all sublists of a given list L.

2. Design a function **reduce** that takes a binary associative operator (function) $f :' a * 'a \rightarrow' a$ and a list $[a_0, a_1, ..a_{n-1}] : 'a\ list$, $n > 0$ and applies this function as a whole on the list. Use this function to define the following:

   (a) The maximum of a list of reals.

   (b) The concatenation of a list of characters (the function implode).

   (c) The median element in a list of integers

   (d) The mean of a list of reals.

   In which of the above cases is it necessary to raise an exception in case the list is empty? What should be done in the other cases?

3. Design a function **filter** that takes a predicate p and a list L and produces the list of elements of L that satisfy the predicate P. What should be the type of predicate p to make this function work? Use this function to define the following:

   (a) Find those elements of a list of strings that begin with the character #"a".

   (b) Find those elements of a list of strings that are atmost 3 characters long.

   (c) Determine the list of all primes in the list generated by the function $AS4(3, 2, n)$ for an arbitrary large value of n.

4. Find out what do the functions $foldl$, $foldr$ and $exists$ do. Implement these functions yourself. As you may find out, $foldl$ and $foldr$ perform the same task for many binary operators.

   (a) So what is the basic difference between $foldl$ and $foldr$? Give an an example of a binary operator and a list argument for which $foldl$ and $foldr$ yield different results.

(b) Is there any difference between their time complexities?

5. Matrices can be implemented as Lists of Lists in SML. Write the following polymorphic functions that will work for a general matrix (like matrix of reals, booleans etc):

   (a) a function *polyAdd* that can add matrices. This function will need an additional argument apart from the input matrices: a function *plus* that performs addition on the corresponding elements. For example, simple addition for reals and OR for booleans may be used as the input function. This function should raise exceptions for various situations.

   (b) a function *polyScalarMult* which multiplies each element of a given matrix by a single element. This requires a binary operation *mult* on the type of elements that make up the matrix.

   (c) a function *polyMult* that can multiply two general matrices. This function will need two additional arguments apart from the input matrices: a binary function *plus* and another binary function *mult* that performs multiplication on the elements of the 2 matrices. For example, for real matrices, you may use simple addition and multiplication as argument functions and for booleans you can use OR for addition and AND for multiplication. The function shoould raise an exception InvalidInput if matrices cannot be multiplied or we do not have a valid 2D matrix as input. Analyze the time complexity of your program? Would the code have been simpler if we only had say a matrix of reals?

   (d) a function *isEqual* that given a function that determines element equality and two matrices, returns true if the two matrices are the same.

6. Design a recursive data type `nat` for Natural Numbers (do not use any built-in data type in the definition of `nat`). Note that we will have natural numbers starting with 0, so have your base case designed accordingly. Implement the following functions using this data type:

   (a) $isZero : nat \rightarrow bool$ that checks whether the natural number is zero or not.

   (b) $pred : nat \rightarrow nat$ that finds the predecessor of a natural number. It should also raise an exception if the argument is zero.

   (c) $add : nat \rightarrow nat$ that adds two natural numbers.

   (d) $toInt : nat \rightarrow int$ that converts natural number to built-in integer.

   (e) $fromInt : int \rightarrow nat$ that converts an integer to natural number.