Introduction to Computer Science
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: CH-232-A
Date: 2019-11-08
Due: 2019-11-15

ICS 2019 Problem Sheet #9

Problem 9.1: *full adder using different kinds of gates* (1+1+1+1 = 4 points)

A full adder digital circuit was introduced in class. It is defined by the following two boolean functions:

$$\begin{aligned} S &= A\dot{\vee}B\dot{\vee}C_{in} \\ C_{out} &= (A \wedge B) \vee (C_{in} \wedge (A\dot{\vee}B)) \end{aligned}$$

a) Write both functions as a disjunction of product terms.

b) Write both functions as a conjunction of sum terms.

c) Write both functions using only not (¬) and not-and (↑) operations.

d) In a digital circuit, we can easily reuse common terms. Draw a small digital circuit implementing S and C_{out} using NAND gates only.

Problem 9.2: *fold function duality theorems* (2+2+2 = 6 points)

The fold functions compute a value over a list by applying an operator to the list elements and a neutral element. The foldl function assumes that the operator is left associative, the foldr function assumes that the operatore is right associative. For example, the function call

```
foldl (+) 0 [3,5,2,1]
```

results in the computation of ((((0+3)+5)+2)+1) and the function call

```
foldr (+) 0 [3,5,2,1]
```

results in the computation of (3+(5+(2+(1+0)))). The value computed by the fold functions may be more complex than a simple scalar. It is very well possible to construct a new list as part of the fold. For example:

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs
```

The evaluation of `map' (+3) [1,2,3]` results in the list `[4,5,6]`. There are several duality theorems that can be stated for the fold functions. Prove the following three duality theorems:

a) Let `op` be an associative operation with `e` as the neutral element:

   `op` is associative: `(x op y) op z = x op (y op z)`
   `e` is neutral element: `e op x = x` and `x op e = x`

   Then the following holds for finite lists `xs`:

   ```
   foldr op e xs = foldl op e xs
   ```

b) Let `op1` and `op2` be two operations for which

   ```
   x `op1` (y `op2` z) = (x `op1` y) `op2` z
   x `op1` e           = e `op2` x
   ```

holds. Then the following holds for finite lists `xs`:

```
foldr op1 e xs = foldl op2 e xs
```

c) Let `op` be an associative operation and `xs` a finite list. Then

```
foldr op a xs = foldl op' a (reverse xs)
```

holds with

```
x op' y = y op x
```