ICS 2022 Problem Sheet #9

**Problem 9.1:** JK flip-flops

JK flip-flops, also colloquially known as jump/kill flip-flops, augment the behaviour of SR flip-flops. The letters J and K were presumably picked by Eldred Nelson in a patent application.

The sequential digital circuit shown below shows the design of a JK flip-flop based on two SR NAND latches. Assume the circuit’s output is Q = 0 and that the inputs are J = 0 and K = 0, and that the clock input is C = 0. (You can make use of the fact that we already know how an SR NAND latch behaves.)

J



Q

C

Q

K

1. Suppose J transitions to 1 and C transitions to 1 soon after. Create a copy of the drawing and indicate for each line whether it carries a 0 or a 1.



1

1

1

0

1

1

0

1

J

0

1

0

Q

1

1

C

1

0

1

0

1

0

1

0

1

Q

0

K

0

0

0



1

1. Some time later, C transitions back to 0 and soon after J transitions to 0 as well. Create another copy of the drawing and indicate for each line whether it carries a 0 or a 1

0000

1

1

1

0

1



1

0

J

0

0

1

Q

0

0

C

1

1

0

0

0

1

1

0

0

Q

0

K

1

1

1

0

1. Some time later, J and K both transition to 1 and C transitions to 1 soon after. Create another copy of the drawing and indicate for each line whether it carries a 0 or a 1.

0

0

1

0

1

0



1

1

J

1

0

1

Q

1

1

C

0

1

1

0

0

1

0

1

1

Q

1

K

1

0

1

1. Finally, C transitions back to 0 and soon after J and K both transition to 0 as well. Create another copy of the drawing and indicate for each line whether it carries a 0 or a 1.

111



0

0000

0

0

1

1

1

J

1

1

0

Q

0

0

C

0

0

0

1

1

1

1

0

0

Q

1

K

0

0

1

0

**Problem 9.2:** fold function duality theorems

The fold functions compute a value over a list (or some other type that is foldable) 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 operator is right associative. For example, the function application

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

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

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:

1 map' :: (a -> b) -> [a] -> [b]

2 map' f xs = foldr ((:) . f) [] xs

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

1. 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 let xs = [1,2,3]

( 1 op ( 2 op ( 3 op e))) = ((( e op 1) op 2) op 3)

( 1 op ( 2 op ( e op 3))) = ((( e op 1) op 2) op 3) (e is neutral element)

(( 1 op ( 2 op e) op 3)) = ((( e op 1) op 2) op 3) (op is associative)

(( 1 op ( e op 2) op 3)) = ((( e op 1) op 2) op 3) (e is neutral element)

((( 1 op e) op 2) op 3) = ((( e op 1) op 2) op 3) (op is associative)

**((( e op 1) op 2) op 3) = ((( e op 1) op 2) op 3)** (e is neutral element)

foldr op e xs = foldl op e xs holds for finite lists xs

1. Let op1 and op2 be two operations for which holds.

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

x `op1` e = e `op2` x

Then the following holds for finite lists xs:

foldr op1 e xs = foldl op2 e xs let xs = [1,2,3]

( 1 `op1` ( 2 `op1` ( 3 `op1` e))) = ((( e `op2` 1) `op2` 2) `op2` 3)

( 1 `op1` ( 2 `op1` ( e `op2` 3))) = ((( e `op2` 1) `op2` 2) `op2` 3) (x `op1` e = e `op2` x)

(( 1 `op1` ( 2 `op1` e) `op2` 3)) = ((( e `op2` 1) `op2` 2) `op2` 3) (x `op1` (y `op2` z) = (x `op1` y) `op2` z)

(( 1 `op1` ( e `op2` 2) `op2` 3)) = ((( e `op2` 1) `op2` 2) `op2` 3) (x `op1` e = e `op2` x)

((( 1 `op1` e) `op2` 2) `op2` 3) = ((( e `op2` 1) `op2` 2) `op2` 3) (x `op1` (y `op2` z) = (x `op1` y) `op2` z)

**((( e `op2` 1) `op2` 2) `op2` 3) = ((( e `op2` 1) `op2` 2) `op2` 3)** (x `op1` e = e `op2` x)

foldr op1 e xs = foldl op2 e xs holds for finite lists xs

1. Let op be an associative operation and xs a finite list. (op is associative: (x op y) op z = x op (y op z))

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

Then holds with

x op' y = y op x

foldr op a xs = foldl op' a (reverse xs) let xs = [1,2]

foldr op a [1,2] = foldl op' a [2,1]

(1 op ( 2 op a)) = (( a op’ 2) op’ 1)

((1 op 2) op a) = ( a op’ (2 op’ 1))

Which is true 1. 1 op 2 = 2 op’ 1 = b x op' y = y op x

2. b op a = a op’ b x op' y = y op x

**Then this holds that if foldr op a xs = foldl op' a (reverse xs) then it follows that x op' y = y op x.**