Intro to Computer Science Assignment 2

Yiping Deng

2017-09-20

6.1 Giving two boolean variable P, Q
By truth table, we have $P \to Q = \neg P \vee Q$

$$P \vee Q = \neg\neg P \vee Q = \neg P \to Q$$
$$P \wedge Q = \neg(\neg P \vee \neg Q) = \neg(\neg\neg P \to \neg Q) = \neg(P \to \neg Q)$$

6.2

1) Truth table proof:

P	Q	R	S	(not P or Q) and (not Q or R) and (not R or S) and (not S or P)
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

P	Q	R	S	not (P or Q or R or S) or (P and Q and R and S)
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

So, 2

2) DNF form:

$$
\begin{aligned}
\varphi(M, N, P, Q, R, S) = & \\
& = ((\neg(P \vee Q \vee R \vee S)) \vee (P \wedge Q \wedge \wedge R \wedge S))) \wedge M \wedge \neg N \\
& = (\neg P \wedge \neg Q \wedge \neg R \wedge \neg S \wedge M \wedge \neg N) \vee (P \wedge Q \wedge R \wedge S \wedge M \wedge \neg N)
\end{aligned}
$$

3)

$$
\begin{aligned}
\varphi(M,N,P,Q,R,S) &= (\neg P \vee Q) \wedge (\neg Q \vee R) \wedge (\neg R \vee S) \wedge (\neg S \vee P) \wedge M \wedge \neg N \\
(\neg P \vee Q) \wedge (\neg Q \vee R) &= ((\neg P \vee Q) \wedge \neg Q) \vee ((\neg P \vee Q) \wedge R) \\
&= (\neg P \wedge \neg Q) \vee (Q \wedge \neg Q) \vee ((\neg P \vee Q) \wedge R) \\
&= (\neg P \wedge \neg Q) \vee 0 \vee ((\neg P \vee Q) \wedge R) \\
&= \neg (P \vee Q) \vee ((\neg P \vee Q) \wedge R) \\
(\neg R \vee S) \wedge (\neg S \vee P) &= \neg (R \vee S) \vee ((\neg R \vee S) \wedge P) \\
& (\neg (R \vee S) \vee ((\neg R \vee S) \wedge P)) \wedge (\neg (P \vee Q) \vee ((\neg P \vee Q) \wedge R)) \\
&= (\neg (P \vee Q \vee R \vee S) \\
& \vee ((\neg R \vee S) \wedge P) \wedge \neg (P \vee Q)) \\
& \vee (\neg (R \vee S) \wedge ((\neg P \vee Q) \wedge R)) \\
& \vee ((\neg R \vee S) \wedge P) \wedge ((\neg P \vee Q) \wedge R)) \\
&= (\neg (P \vee Q \vee R \vee S) \\
& \vee ((\neg R \vee S) \wedge P) \wedge ((\neg P \vee Q) \wedge R)) \\
&= (\neg (P \vee Q \vee R \vee S) \\
& \vee ((\neg R \vee S) \wedge (\neg P \vee Q) \wedge R \wedge P)) \\
&= (\neg (P \vee Q \vee R \vee S) \vee (P \wedge Q \wedge R \wedge S) \\
\varphi(M,N,P,Q,R,S) &= (\neg (R \vee S) \vee ((\neg R \vee S) \wedge P)) \wedge (\neg (P \vee Q) \vee ((\neg P \vee Q) \wedge R)) \wedge M \wedge \neg N \\
&= ((\neg (P \vee Q \vee R \vee S)) \vee (P \wedge Q \wedge \wedge R \wedge S))) \wedge M \wedge \neg N \\
&= (\neg P \wedge \neg Q \wedge \neg R \wedge \neg S \wedge M \wedge \neg N) \vee (P \wedge Q \wedge R \wedge S \wedge M \wedge \neg N)
\end{aligned}
$$

6.3

```
-- Define a data type that behaves like a tuple
data Move = MakeMove Int Int
instance Show Move where
    show (MakeMove x y) = show (x,y)

solveHanoi :: Int -> Int -> Int -> [Move]
-- the first argument is the number of element to move
-- the second is the source of the element,in 1 to 3
-- the third is the destination of the element, in 1 to 3
-- we should be able to satisfies all the courner cases
-- if it is one to move, make one move
-- if it is zero, give a empty lists
-- recursive case study:
--
--  recursive case assumption:
--      when moving, only the source is occupied, the destination and the auxillary heap
--      is empty
```

```haskell
--  divide: break the case that move x  from source to destination into:
--      1. move (x - 1) from source to the auxillary(e.g.: if source is the first heap,
--      the destination is the second heap, the auxillary is the third one)
--      2. move the remaining one, which is the biggest, from the source,
--      to the destination, which is empty right now.
--      3. move (x - 1) elements in step 1)
--      from auxillary to the destination
--  conquer:
--      solve the subproblem step 1) and step 3) recursively.
--      note: after step 1), we actually breaks
--      the recursive assumption.( there is one element
--      in the source) step 2) recover this
--      state(it is non-recursive), then we can safely recursively
--      solve step 3) recursively
--  combine:
--      to get the result, we simply concatenate the 3 lists
solveHanoi 0 _ _ = []
solveHanoi 1 x y = [MakeMove x y]
solveHanoi x 1 3 = solveHanoi (x - 1) 1 2 ++ (solveHanoi 1 1 3) ++ (solveHanoi (x - 1) 2 3)
solveHanoi x 3 1 = solveHanoi (x - 1) 3 2 ++ (solveHanoi 1 3 1) ++ (solveHanoi (x - 1) 2 1)
solveHanoi x 1 2 = solveHanoi (x - 1) 1 3 ++ (solveHanoi 1 1 2) ++ (solveHanoi (x - 1) 3 2)
solveHanoi x 2 1 = solveHanoi (x - 1) 2 3 ++ (solveHanoi 1 2 1) ++ (solveHanoi (x - 1) 3 1)
solveHanoi x 3 2 = solveHanoi (x - 1) 3 1 ++ (solveHanoi 1 3 2) ++ (solveHanoi (x - 1) 1 2)
solveHanoi x 2 3 = solveHanoi (x - 1) 2 1 ++ (solveHanoi 1 2 3) ++ (solveHanoi (x - 1) 1 3)

hanoi :: Int -> [Move]
hanoi x = solveHanoi x 1 3
```