Code (Wednesday Week 4)

Editor

Haskell

```haskell
import Test.QuickCheck

data Abstract = A { text :: String, cursor :: Int }
   deriving (Show, Eq)

-- don't worry about this too much for now
instance Arbitrary Abstract where
  arbitrary = do
    t <- arbitrary
    c <- choose (0, length t)
    pure (A t c)

wellformed :: Abstract -> Bool
wellformed (A t c) = c >= 0 && c <= length t

-- Data Invariant properties
prop_arbitrary_ok a = wellformed a

prop_einit_ok s = wellformed (einitA s)
prop_moveLeft_ok a = wellformed (moveLeftA a)
prop_moveRight_ok a = wellformed (moveRightA a)
prop_insertChar_ok c a = wellformed (insertCharA c a)
prop_deleteChar_ok a = wellformed (deleteCharA a)

-- Abstract Implementation
einitA :: String -> Abstract
einitA s = A s 0

stringOfA :: Abstract -> String
stringOfA (A s c) = s

moveLeftA :: Abstract -> Abstract
moveLeftA (A t c) = A t (max 0 (c-1))

moveRightA :: Abstract -> Abstract
moveRightA (A t c) = A t (min (length t) (c+1))

insertCharA :: Char -> Abstract -> Abstract
insertCharA x (A t c) = let (t1, t2) = splitAt c t
                         in A (t1 ++ [x] ++ t2) (c+1)
```

```haskell
deleteCharA :: Abstract -> Abstract
deleteCharA (A t c) = let (t1, t2) = splitAt c t
                       in A (t1 ++ drop 1 t2) c

data Concrete = C [Char] [Char]
  deriving (Show, Eq)

instance Arbitrary Concrete where
  arbitrary = C <$> arbitrary <*> arbitrary

toAbstract :: Concrete -> Abstract
toAbstract (C ls rs) = A (reverse ls ++ rs) (length ls)

-- Data Refinement Properties
prop_init_r s =
    toAbstract (einit s) == (einitA s)
prop_stringOf_r c =
    stringOf c == stringOfA (toAbstract c)
prop_moveLeft_r c =
    toAbstract (moveLeft c) == moveLeftA (toAbstract c)
prop_moveRight_r c =
    toAbstract (moveRight c) == moveRightA (toAbstract c)
prop_insertChar_r x c =
    toAbstract (insertChar x c) == insertCharA x (toAbstract c)
prop_deleteChar_r c =
    toAbstract (deleteChar c) == deleteCharA (toAbstract c)
-- Concrete Implementation
einit :: String -> Concrete
einit s = C [] s
stringOf :: Concrete -> String
stringOf (C ls rs) = reverse ls ++ rs

moveLeft :: Concrete -> Concrete
moveLeft (C (l:ls) rs) = C ls (l:rs)
moveLeft c = c

moveRight :: Concrete -> Concrete
moveRight (C ls (r:rs)) = C (r:ls) rs
moveRight c = c

insertChar :: Char -> Concrete -> Concrete
insertChar x (C ls rs) = C (x: ls) rs

deleteChar :: Concrete -> Concrete
deleteChar (C ls (_:rs)) = C ls rs
deleteChar c = c
```