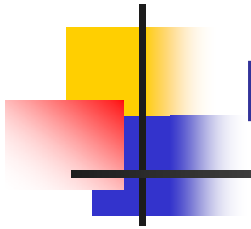
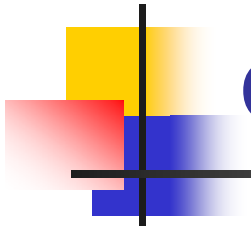
# Quickcheck – Spezifikationsbasiertes Testen von Haskellprogrammen

Koen Claessen, John Hughes



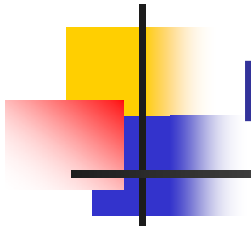
# Motivation

- Dijkstra: „Testen kann nie das Fehlen von Fehlern zeigen, sondern lediglich vorhandene Fehler aufdecken.“
- Quickcheck
  - erlaubt systematisches Testen von Haskellprogrammen
  - liefert keine Korrektheitsnachweise
  - erhöht nur das Vertrauen in Programme



# Quickcheck

- definiert eine formale Spezifikationssprache
  - um zu testende Eigenschaften direkt im Source-Code auszudrücken
- definiert eine Testdaten-Generierungssprache
  - um große Testmengen kompakt zu beschreiben
- umfasst ein Werkzeug
  - um die spezifizierten Eigenschaften zu testen und eventuell entdeckte Fehler zu melden
- wird mit Hugs und GHC ausgeliefert



# Einfache Beispiele

**Eigenschaften sind monomorphe Boolesche Funktionen.**

- **einfach:**

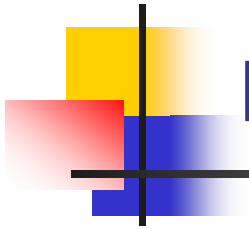
```
prop_PlusAssociative :: Int -> Int -> Int -> Bool
prop_PlusAssociative x y z = (x+y)+z == x+(y+z)
```

- **bedingt:**

```
prop_InsertOrdered :: Int -> [Int] -> Property
prop_InsertOrdered x xs
       = ordered xs ==> ordered (insert x xs)
```

- **allquantifiziert:**

```
prop_InsertOrdered2 :: Int -> Property
prop_InsertOrdered2 x
       = forAll orderedLists $
                  \ xs -> ordered (insert x xs)
```



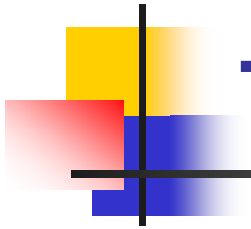
# Fallstudie: Warteschlangen

```
emptyQueue   :: Queue a
enqueue      :: Queue a -> a -> Queue a
dequeue      :: Queue a -> Queue a
firstQueue   :: Queue a -> a
isEmptyQueue :: Queue a -> Bool

Gleichungsspezifikation:
------------------------
dequeue (enqueue emptyQueue x) = emptyQueue
isEmptyQueue q = False
   -> dequeue (enqueue q x) = enqueue (dequeue q) x

firstQueue (enqueue emptyQueue x) = x
isEmptyQueue q = False
   -> firstQueue (enqueue q x) = firstQueue q

isEmptyQueue emptyQueue    = True
isEmptyQueue (enqueue q x) = False
```



# Testgeneratoren

- erzeugen Testdaten -> Monadentyp Gen a
- vordefiniert für die meisten Typen, modifizierbar/erweiterbar -> Typklasse Arbitrary

```
class Arbitrary a where arbitrary :: Gen a
```

- eigene Testgeneratoren mit forAll nutzbar
  - `forAll :: Gen a -> (a -> Bool) -> Property`
- Auswahlfunktionen
  - `choose    :: Random a => (a,a) -> Gen a`
  - `oneof     :: [ Gen a] -> Gen a`
  - `frequency :: [(Int, Gen a)] -> Gen a`
- Größenkontrolle
  - `sized     :: (Int -> Gen a) -> Gen a`
  - `resize    :: Int -> Gen a -> Gen a`

# Beispiel: Testgenerator orderedLists

```
orderedLists  = do  x <- arbitrary
                    listsFrom x
 where
  listsFrom x = oneof [ return [],
                        do  y <- atLeast x
                            liftM (x:) (listsFrom y) ]
  atLeast x   = do  diff <- arbitrary
                    return (x + abs diff)
```

**alternative Definition von `listsfrom`:**

```
  listsFrom x = frequency
                  [ (1, return []),
                    (4, do  y <- atLeast x
                            liftM (x:) (listsFrom y))]
```

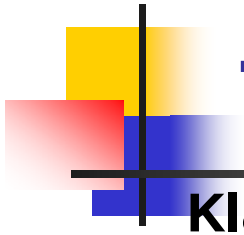
# Beispiel: Testgenerator für Bäume

```
data Tree = Leaf Int | Branch Tree Tree
tree ::   Gen Tree
tree =    oneof [liftM Leaf arbitrary,
                 liftM2 Branch tree tree]
```

**Problem: Baumerzeugung muss nicht unbedingt terminieren.**

**-> Größenkontrolle:**

```
tree = sized stree
stree :: Int -> Gen Tree
stree 0 = liftM Leaf arbitrary
stree n |n>0 =   oneof [liftM Leaf arbitrary,
                        liftM2 Branch subt subt]
     where subt = stree (n `div` 2)
```



# Testkontrolle

**Klassifikation von Testfällen:**

- `trivial :: Bool -> Property –> Property`

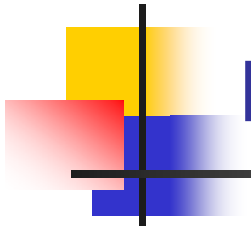
  **Beispiel:**

```
prop_InsertOrdered x xs
      = ordered xs =>
        (trivial (length xs <= 2) $
                  ordered(insert x xs)
```

- `classify :: Property -> String -> Property`

  **Beispiel:**

```
prop_InsertOrdered x xs
      = ordered xs =>
        (classify (null xs) „empty“ $
        (classify (length xs == 1) „unit“ $
                  ordered(insert x xs)
```



# Fazit

- leichte Überprüfung von Spezifikationen und Implementierungen durch systematisches Testen
- kein Korrektheitsnachweis, aber viele Fehler oder Inkonsistenzen werden aufgedeckt
- flexibel durch Möglichkeit der Testgenerierung und -kontrolle