Typklassen

Philip Wadler

Typklassen

QuickSort-Algorithmus: prinzipiell für alle Element-Typen

- für die totale Ordnung $\leq$ definiert

Umsetzung in Haskell:

1. Versuch: eigene Funktion je Typ

```
qsortI :: [Integer] -> [Integer]
qsortI []      = []
qsortI (p:ps) =      (qsortI (filter (\x -> x<=p) ps))
                ++ p:(qsortI (filter (\x -> x> p) ps))

qsortD :: [Double] -> [Double]
qsortD []      = []
qsortD (p:ps) =      (qsortD (filter (\x -> x<=p) ps))
                ++ p:(qsortD (filter (\x -> x> p) ps))
...
```

Softwaretechnisch katastrophal:

- Code-Duplizierung, nicht anwendbar auf neue Typen

Typklassen

QuickSort-Algorithmus: prinzipiell für alle Element-Typen

- für die totale Ordnung $\leq$ definiert

Umsetzung in Haskell:

2. Versuch: als polymorphe Funktion

```
qsort :: [t] -> [t]
qsort []     = []
qsort (p:ps) =      (qsort (filter (\x -> x<=p) ps))
               ++ p:(qsort (filter (\x -> x> p) ps))
```

Funktioniert nicht:

- nicht für alle Typen `t` existieren Ordnungsfunktionen $<=$ und $>$

Typklassen

QuickSort-Algorithmus: prinzipiell für alle Element-Typen

- für die totale Ordnung $\leq$ definiert

Umsetzung in Haskell:

Lösung: als polymorphe Funktion mit Typeinschränkung

```
qsort :: Ord t => [t] -> [t]
qsort []     = []
qsort (p:ps) =      (qsort (filter (\x -> x<=p) ps))
                 ++ p:(qsort (filter (\x -> x> p) ps))
```

Zu lesen: Für alle Instanzen `t` der Typklasse **`Ord`** ist `qsort` vom Typ `[t] -> [t]`

- Instanzen `t` von **`Ord`** implementieren `<=,<,>,>=,...`
- Schreibe: "**`Ord`** `t`" falls `t` Instanz von **`Ord`**

  `Ord Integer`, **`Ord Double`**, ...

Standard-Typklassen

Typklassen

- Fassen Typen anhand auf ihnen definierter Operationen zusammen
- Möglichst zu geltende Gesetze: Dokumentation
- Grob: Ähneln prinzipiell Java Interfaces
- Grob: Typparameter `t` in `C t` entspricht dem Typ von **`this`** in Java

Klasse: Typen mit Gleichheit `Eq t`

Operationen:
```
(==) :: t -> t -> Bool
(/=) :: t -> t -> Bool
```

Gesetze: `(==)` berechnet Äquivalenzrelation
```
(/=) = (\x y -> not (x == y))
```

Instanzen: `Eq Integer, Eq Char, Eq Bool, ...`

`Eq [t]` — aber nur, falls `(Eq t)`

Aber nicht: `(s->t)`

Standard-Typklassen

Typklassen

- Fassen Typen anhand auf ihnen definierter Operationen zusammen
- Möglichst zu geltende Gesetze: Dokumentation
- Grob: Ähneln prinzipiell Java Interfaces
- Grob: Typparameter `t` in `C t` entspricht dem Typ von **`this`** in Java

Klasse: Geordnete Typen `Ord t`

Operationen:
```
(<=) :: t -> t -> Bool
(<)  :: t -> t -> Bool
...
```

Gesetze: Operator `(<)` berechnet totale Ordnung
Operatoren `(<=)`, `(>)` ... sind kompatibel mit `(<)` und `==`

Instanzen: `Ord Float`, `Ord Integer`, `Ord Char`, ...
Aber nicht: `(s->t)`

Standard-Typklassen

Typklassen

- Fassen Typen anhand auf ihnen definierter Operationen zusammen
- Möglichst zu geltende Gesetze: Dokumentation
- Grob: Ähneln prinzipiell Java Interfaces
- Grob: Typparameter `t` in `C t` entspricht dem Typ von **`this`** in Java

Klasse: Numerische Typen **`Num`** `t`

Operationen:

```
(+) :: t -> t -> t          negate :: t -> t
(*) :: t -> t -> t          abs :: t -> t
(-) :: t -> t -> t          signum :: t ->t
fromInteger :: Integer -> t
```

Gesetze: Assoziativität, Kommutativität, Distributivität, . . .
`fromInteger 1` und `fromInteger 0` sind neutral bzgl. `(*)`, `(+)`

Instanzen: **`Num Float`**, **`Num Integer`**, . . .

Standard-Typklassen

Typklassen

- Fassen Typen anhand auf ihnen definierter Operationen zusammen
- Möglichst zu geltende Gesetze: Dokumentation
- Grob: Ähneln prinzipiell Java Interfaces
- Grob: Typparameter `t` in `C t` entspricht dem Typ von **`this`** in Java

Klasse: Anzeigbare Typen **`Show`** `t`

Operationen: `show :: t -> String`

Gesetze: —

Instanzen: **`Show Float`**, **`Show Bool`**, ...

Standard-Typklassen

Typklassen

- Fassen Typen anhand auf ihnen definierter Operationen zusammen
- Möglichst zu geltende Gesetze: Dokumentation
- Grob: Ähneln prinzipiell Java Interfaces
- Grob: Typparameter `t` in `C t` entspricht dem Typ von **`this`** in Java

Klasse: Aufzählungstypen **`Enum`** `t`

Operationen:

```
succ :: t -> t              pred :: t -> t
toEnum :: Int -> t          fromEnum :: t -> Int
enumFromTo :: t -> t -> [t]
```

Instanzen: **`Enum Bool`**, **`Enum Int`**, **`Enum Char`** ...

Notation: `[a..b]` ≡ `enumFromTo a b`

Typklassendefinition

Karlsruher Institut für Technologie

Typklassen-Definition: Eq t

```
class (Eq t) where
  (==) :: t -> t -> Bool
  (/=) :: t -> t -> Bool
```

Typklassen-Instanziierung: Gleichheit von Bool

```
instance (Eq Bool) where
  True  == True  = True
  False == False = True
  False == True  = False
  True  == False = False
```

```
  True  /= True  = False
  False /= False = False
  False /= True  = True
  True  /= False = True
```

Typklassendefinition

Typklassen-Definition: **`Eq`** `t` mit Default-Implementierungen

```
class (Eq t) where
  (==) :: t -> t -> Bool
  (/=) :: t -> t -> Bool

  x /= y  = not (x == y)
  x == y  = not (x /= y)
```

Typklassen-Instanziierung: Gleichheit von **`Bool`**

```
instance (Eq Bool) where
  True  == True  = True
  False == False = True
  False == True  = False
  True  == False = False
```

oder

```
instance (Eq Bool) where
  True  /= True  = False
  False /= False = False
  False /= True  = True
  True  /= False = True
```

- Fehlende Implementierungen: Default-Implementierung

$\Rightarrow$ $\{==\}$ und $\{/=\}$ sind je minimal-vollständig

Automatische Instanziierung

Karlsruher Institut für Technologie

Gleichheit für Datentypen: `(Eq Shape)`

```
data Shape =   Circle Double -- Radius
             | Rectangle Double Double -- Seitenlängen
             | Square Double     deriving Eq
```

Automatische Instanziierung: `deriving Eq`

- verschiedene Konstruktoren $\Rightarrow$ verschiedene Werte
- gleicher Konstruktor, verschiedene Parameter $\Rightarrow$ verschiedene Werte
- Sowieso: gleicher Konstruktor, gleiche Parameter $\Rightarrow$ gleicher Wert

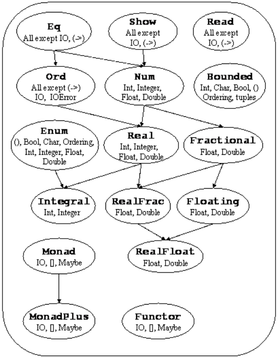
```
Circle 1 == Square 1          ⇒   False
Circle 1 == Circle 3          ⇒   False
Rectangle 1 1 == Square 1     ⇒   False
Square 2 == Square 2          ⇒   True
```

Automatische Instanziierung: Auch für `Show`, `Ord`, `Enum`

Vererbung von Typklassen

Typklassen-Hierarchie: Vererbung

```
data Ordering = LT | EQ | GT
class  (Eq t) => Ord t   where
    compare  :: t -> t -> Ordering
    (<), (<=), (>), (>=) :: t -> t -> Bool

    compare x y
      | x == y     = EQ
      | x <= y     = LT
      | otherwise = GT

    x < y  = lt (compare x y)
      where lt LT = True
            lt x  = False

    x <= y = ...
```



Standard-Typklassen

- Jede Instanz von `Ord` auch Instanz von `Eq`

$\Rightarrow$ `(==) :: t -> t -> Bool` verfügbar für Default-Implementierung

Minimal-Vollständing: {`compare`} und {`<=`}

Generische Instanziierung

Instanziierung: `Ord (s,t)` von beliebigen Tupel-Typen

- Möglich, falls `Ord t` und `Ord s`
- Ordnung: erst nach `s`, dann nach `t`

```
instance (Eq s,Eq t) => Eq (s,t) where
    (a,b) == (a',b') = (a==a') && (b==b')

instance (Ord s,Ord t) => Ord (s,t) where
    (a,b) <= (a',b') = (a<=a') && (b<=b')
```

Funktoren

Funktor

- Abbildung `f` von Typen auf Typen
- Zusammen mit Funktion

```
m :: (s -> t) -> (f s) -> (f t)
```

sodass

```
m id     = id
m (f.g)  = (m f) . (m g)
```

Beispiel in Haskell: Listen `[t]`

- `f` = Listen-Typ-Konstruktor `[]`
- `m` = `map :: (s -> t) -> [s] -> [t]`

Funktoren

Funktor

- Abbildung `f` von Typen auf Typen
- Zusammen mit Funktion

```
m :: (s -> t) -> (f s) -> (f t)
```

sodass

```
m id     = id
m (f.g)  = (m f) . (m g)
```

Als Haskell-Typklasse: Listen `[t]`

```haskell
class Functor f where
  fmap :: (s -> t) -> (f s) -> (f t)


instance Functor [] where
  fmap = map
```

Funktoren

Funktor

- Abbildung `f` von Typen auf Typen
- Zusammen mit Funktion

```
m :: (s -> t) -> (f s) -> (f t)
```

sodass

```
m id     = id
m (f.g)  = (m f) . (m g)
```

Als Haskell-Typklasse: Bäume `Tree t`

```
class Functor f where
  fmap :: (s -> t) -> (f s) -> (f t)

data Tree t =  Leaf | Node (Tree t) t (Tree t)
instance Functor Tree where
  fmap f Leaf = Leaf
  fmap f (Node left x right) =
      Node (fmap f left) (f x) (fmap f right)
```

Vergleich: Typklassen/Java

Haskell Typklassen	Java

`class Show t where`
`  show :: t -> String`
	`interface Show {`
	`  String show();`
	`}`

`class  Show t => B t  where`
`   ...`
	`interface B extends Show {`
	`   ...`
	`}`

`class Coll c where`
`  contains :: (Ord t) =>`
`      (c t) -> t -> Bool`
	`interface Coll<T extends Comparable<T>> {`
	`  Boolean contains(T t);`
	`}`

`class A t where`
`  foo :: t -> t`
`  bar :: t -> Bool`
`  bar x = True`
	`abstract class A {`
	`  abstract A foo();`
	`  Boolean bar() {`
	`    return true;`
	`  }`
	`}`

`instance (Show Int) where`
`  show x = ...`
	`class Integer implements Show {`
	`  String show() {`
	`    return ...;`
	`  }`
	`}`

Physikalische Größen

Quelle von Programmfehler: Verwechselung physikalischer Einheiten

- Mars Climate Orbiter: Pound-Force statt Newton
  $1 lb_f \approx 4,448 N$
- Verlust der Sonde, Projektkosten: $\approx$ 327 Millionen $

Abhilfe in Haskell: Verwechselungen statisch ausschließen
Einheiten automatisch konvertieren

- Ausdruck: $10 lb_f + 46 N$ ergibt Typfehler
- Keine Laufzeitkosten für Einheitenprüfung
- Konvertierung: $10 lb_f +_{force} 46 N \Rightarrow^+ \; 90.48 N$
- Umsetzung: Algebraische Datentypen, Typklassen

Physikalische Größen

Physikalische Größe: Zahl, Maßeinheit

- $11m$, $10yd$, $130\frac{km}{h}$, ...
- Zahlen als Haskell-Wert, Einheiten als Haskell-Typ

```haskell
data KiloMetrePerHour = Kph Double
data MetrePerSecond = Mps Double

advised :: KiloMetrePerHour
advised = Kph 130.0

c :: MetrePerSecond
c = Mps 299792458
```

```haskell
data Metre = M Double
data Yard = Yd Double

endzone :: Yard
endzone = Yd 10

penalty :: Metre
penalty = M 11.0
```

Rechnen mit Physikalische Größen

Rechnoperationen: $+$, $-$, sowie $\times$ mit Zahlen

- Einheiten bleiben Erhalten

```
class Unit u where
  plus    :: u -> u -> u
  minus   :: u -> u -> u
  ntimes  :: Double -> u -> u

instance Unit Metre where
   (M x)  `plus`   (M y) = M (x+y)
   (M x)  `minus`  (M y) = M (x-y)
   x      `ntimes` (M y) = M (x*y)

instance Unit Yard where
   (Yd x) `plus`  (Yd y)  = Yd (x+y)
   ...
```

Anwendung: Korrekt/Fehlerhaft

```
   penalty `plus` (M 10)
⇒+ Number 21.0 :: Metre

   10 `ntimes` endzone
⇒+ Yd 100.0 :: Yard
```

```
penalty `plus` endzone

Couldn't match expected
type 'Metre' against inferred
type 'Yard'
```

Konvertierung Physikalischer Größen

Konvertierung: `Metre`, `Yard` verschieden, aber:

- Beides Längeneinheiten $\Rightarrow$ konvertierbar
- Umsetzung: Typklassen

```
class (Unit u) => Length u where
  toMetre   :: u     -> Metre
  fromMetre :: Metre -> u
```

```
instance Length Metre where
  toMetre   = id
  fromMetre = id
```

```
instance Length Yard where
  toMetre   (Yd x) = M  (x*0.9144)
  fromMetre (M x)  = Yd (x/0.9144)
```

Beispiel Konvertierung:

```
  penalty' :: Yard
  penalty' = fromMetre penalty
```

$\Rightarrow^+$ `Yd 12.029746`

Konvertierung Physikalischer Größen

Konvertierung: `Metre`, `Yard` verschieden, aber:

- Beides Längeneinheiten ⇒ konvertierbar
- Umsetzung: Typklassen

```
class (Unit u) => Length u where
  toMetre   :: u     -> Metre
  fromMetre :: Metre -> u
```

```
instance Length Metre where
  toMetre   = id
  fromMetre = id
```

```
instance Length Yard where
  toMetre   (Yd x) = M  (x*0.9144)
  fromMetre (M x)  = Yd (x/0.9144)
```

Ergebniseinheit Addition: Meter

```
lplusl :: (Length l, Length l') =>
          l -> l' -> Metre
lplusl a b = (toMetre a) `plus` (toMetre b)
```

```
   x = (penalty `lplusl` endzone)
⇒+ M 20.144 :: Metre
```

Konvertierung Physikalischer Größen

Konvertierung: `Metre`, `Yard` verschieden, aber:

- Beides Längeneinheiten ⇒ konvertierbar
- Umsetzung: Typklassen

```
class (Unit u) => Length u where
  toMetre   :: u     -> Metre
  fromMetre :: Metre -> u

instance Length Metre where
  toMetre   = id
  fromMetre = id

instance Length Yard where
  toMetre   (Yd x) = M  (x*0.9144)
  fromMetre (M x)  = Yd (x/0.9144)
```

Ergebniseinheit Addition: Typ des Ergebnisterms

```
lplusl :: (Length l, Length l', Length l'') =>
           l -> l' -> l''
lplusl a b = fromMetre ((toMetre a) `plus` (toMetre b))
```

```
   x :: Metre
   x = penalty `lplusl` endzone
⇒+ M 20.144 :: Metre
```

```
   x :: Yard
   x = penalty `lplusl` endzone
⇒+ Yd 22.029 :: Yard
```

Physikalische Eigenschaften

Weitere Eigenschaften: Eigene Typklasse

```
class (Unit u) => Speed u where
  toMetrePerSecond   ::  u -> MetrePerSecond
  fromMetrePerSecond ::  MetrePerSecond -> u

instance Unit MetrePerSecond where
  ...

instance Speed MetrePerSecond where
  toMetrePerSecond   = id
  fromMetrePerSecond = id


instance Unit KiloMetrePerHour where
  ...

instance Speed KiloMetrePerHour where
  toMetrePerSecond   (Kph x)  = Mps (x*(1000/(60*60)))
  fromMetrePerSecond (Mps x)  = Kph (x/(1000/(60*60)))
```

Physikalische Eigenschaften

Weitere Eigenschaften: Eigene Typklasse

```
class (Unit u) => Frequency u where
  toHertz   :: u -> Hertz
  fromHertz :: Hertz -> u

instance Unit Hertz where
  ...

instance Frequency Hertz where
  toHertz   = id
  fromHertz = id


instance Unit RevolutionsPerMinute where
  ...

instance Frequency RevolutionsPerMinute where
  toHertz   (Rpm x) = Hz  (x/60)
  fromHertz (Hz x)  = Rpm (x*60)
```

Rechnen mit Physikalische Größen

Rechnoperationen: $\times$, $/$

- Ergebnis: Neue physikalische Eigenschaft
- Länge $\times$ Länge $=$ Fläche
  Geschwindigkeit $/$ Länge $=$ Frequenz

```
sdivl :: (Speed s, Length l, Frequency f) =>
         s -> l -> f
sdivl x y = fromHertz ((toMetrePerSecond x) `div` (toMetre y))
     where div (Mps x) (M y) = Hz (x/y)
```

Wellen-Frequenz: $f = \frac{v}{\lambda}$

```
frequency :: (Length l, Frequency f) => l -> f
frequency lambda = c `sdivl` lambda

red :: Metre            violet :: Metre
red = M 700E-9          violet = M 400E-9
```

Rechnen mit Physikalische Größen

Rechnoperationen: $\times, /$

- Ergebnis: Neue physikalische Eigenschaft
- Länge $\times$ Länge $=$ Fläche
  Geschwindigkeit $/$ Länge $=$ Frequenz

```
sdivl :: (Speed s, Length l, Frequency f) =>
         s -> l -> f
sdivl x y = fromHertz ((toMetrePerSecond x) `div` (toMetre y))
     where div (Mps x) (M y) = Hz (x/y)
```

Vergleichoperationen: $\leq, \geq$

```
instance Ord Metre where              instance Ord Hertz where
    (M x) <= (M y) = x <= y               (Hz x) <= (Hz y) = x <= y

isVisible  :: Metre -> Bool
isVisible lambda = (lambda >= violet) && ( lambda <= red)

isVisible' :: Hertz -> Bool
isVisible' f  = (f <= (frequency violet)) && ( f >= (frequency red))
```

Physikalische Größen: Nachteile

Verbleibende Nachteile der vorgestellten Lösung

- Neuer Datentyp/Instanziierung für jede Einheit
  selbst für Nicht-Basiseinheiten $N = \frac{kg \cdot m}{s^2}$, $W = \frac{kg \cdot m^2}{s^3}$
- Neue Funktion `xmulty`, `xdivy` für jede Kombination (`x`, `y`) physikalischer Eigenschaften

Lösungmöglichkeiten: Mit Haskell-Spracherweiterungen

- Phantom-Typen: Generische $+$, $-$, sowie $\times$ mit Zahlen
- Einheit als Typ-Level Liste:
  $N \simeq [(Kilo, 1_T), (Metre, 1_T), (Second, -2_T)]_T$
- Erfordert Typ-Level Berechnungen:
  $N \times_T Second \Rightarrow_T [(Kilo, 1_T), (Metre, 1_T), (Second, -1_T)]_T$
- z.B.: `http://code.google.com/p/dimensional/`

Validierung mit QuickCheck

Validierung mit QuickCheck

QuickCheck: Haskell-Library zur Generierung von Testdaten

- Vordefinierte Generatoren für Standard-Typen
- Unterstützung bei Definition eigener Generatoren

Spezifikation: als Haskell-Prädikat

```
import Test.QuickCheck

qsortCorrect :: [Integer] -> Bool
qsortCorrect list = ...
```

Valiedierung: mit QuickCheck in `ghci`

QuickCheck-Test

```
*Main> test qsortCorrect
OK, passed 100 tests.
```

mit Testdaten-Anzeige

```
*Main> verboseCheck qsortCorrect
0: []
1: [-5,-1]
...
99: [15,-16,11,10,-14,15,7,-8,1,-4]
OK, passed 100 tests.
```

Spezifikation von Sortieren

QuickSort:

```
qsort []     = []
qsort (p:ps) =       (qsort [x | x <-  ps, x <=  p]
                ++ p: (qsort [x | x <-  ps, x >   p]
```

Spezifikation von QuickSort: Funktionale Korrektheit

1. `qsort list` ist sortiert
2. `qsort list` ist Permutation von `list`

Spezifikation von Sortieren

QuickSort:

```
qsort []     = []
qsort (p:ps) =       (qsort [x | x <-  ps, x <=  p]
               ++ p: (qsort [x | x <-  ps, x >   p]
```

Spezifikation von QuickSort: als Haskell-Prädikat

```
qsortCorrect :: [Integer] -> Bool
qsortCorrect list = let list' = (qsort list)
                    in isSorted list' &&
                       isPerm   list' list
```

Spezifikation von Sortieren

QuickSort:

```
qsort []      = []
qsort (p:ps) =       (qsort [x | x <-  ps, x <=  p]
               ++ p: (qsort [x | x <-  ps, x >   p]
```

Spezifikation von QuickSort: als Haskell-Prädikat

```
qsortCorrect :: [Integer] -> Bool
qsortCorrect list = let list' = (qsort list)
                    in isSorted list' &&
                       isPerm   list' list
```

Sortierung: Eine Liste ist sortiert wenn

- Jedes Element $\leq$ allen Nachfolgern

```
isSorted :: [Integer] -> Bool
isSorted (x:xs) = all (x<=) xs && isSorted xs
isSorted []     = True
```

Spezifikation von Sortieren

QuickSort:

```
qsort []      = []
qsort (p:ps) =        (qsort [x | x <-  ps, x <=  p]
               ++ p: (qsort [x | x <-  ps, x >   p]
```

Spezifikation von QuickSort: als Haskell-Prädikat

```
qsortCorrect :: [Integer] -> Bool
qsortCorrect list = let list' = (qsort list)
                    in isSorted list' &&
                       isPerm   list' list
```

Permutation: Liste `(x:xs)` ist Permutation von Liste `l` wenn

- `x` in `l` enthalten, und
- `xs` ist Permutation von: `l` ohne erstem Vorkommen von `x` in `l`

```
isPerm :: [Integer] -> [Integer] -> Bool
isPerm (x:xs) l = x `elem` l && isPerm xs (delete x l)
isPerm []     []     = True
isPerm []     l      = False
```

Spezifikationen

Spezifikationen: Implementierung

- Unwichtig: Performance
- Wichtig: Müssen selbst korrekt sein
  $\Rightarrow$ Einsatz von Kombinatoren, Comprehensions

Spezifikation...

- ... rein funktionaler Eigenschaften
  z.B. `qsortCorrect`
- ... von Implementierungsdetails
  $\Rightarrow$ Invarianten von Rot-Schwarz-Bäume

Logik mit Rot-Schwarz-Bäumen

n-stelliges `(&&)` und `(||)`

```
andRB :: RedBlackTree Bool -> Bool
andRB = fold (\c l x r -> l && x && r) True

orRB :: RedBlackTree Bool -> Bool
orRB = fold (\c l x r -> l || x || r) False
```

Quantoren $\forall_{x \in \mathtt{tree}} P\ x$ und $\exists_{x \in \mathtt{tree}} P\ x$

```
allRB :: (t -> Bool) -> RedBlackTree t -> Bool
allRB f = andRB . (mapRB (\c x -> f x))

anyRB :: (t -> Bool) -> RedBlackTree t -> Bool
anyRB f = orRB . (mapRB (\c x -> f x))
```

Baum-Mitgliedschaft $\in$

```
elemRB :: (Eq t) => t -> RedBlackTree t -> Bool
elemRB x = anyRB (==x)
```

Logik mit Rot-Schwarz-Bäumen

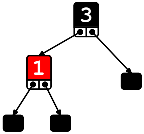
Quantoren $\forall_{\mathtt{t}\ \text{Teilbaum von}\ \mathtt{tree}}\, P\ \mathtt{t}$ und $\exists_{\mathtt{t}\ \text{Teilbaum von}\ \mathtt{tree}}\, P\ \mathtt{t}$

```
subtrees :: RedBlackTree t -> [RedBlackTree t]
subtrees Leaf = [Leaf]
subtrees (Node c left x right) =
  (Node c left x right):(subtrees left)++(subtrees right)


allSubtrees :: (RedBlackTree t -> Bool) -> (RedBlackTree t -> Bool)
allSubtrees pred = (all pred) . subtrees

anySubtree :: (RedBlackTree t -> Bool) -> (RedBlackTree t -> Bool)
anySubtree  pred = (any pred) . subtrees
```

Rot-Schwarz-Bäume

Invarianten:

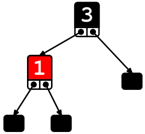
1. Kein roter Knoten hat roten Elternknoten
2. Alle vollständige Pfade haben gleiche Anzahl schwarzer Knoten
3. Baum ist sortiert



Rot-Schwarz-Bäume

Invarianten: als Haskell-Prädikat

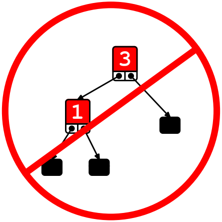
```
rbInvariants :: RedBlackTree Integer -> Bool
rbInvariants tree =
    noRedRed tree &&
    blackSame tree &&
    allSorted tree
```



Rot-Schwarz-Bäume

Invarianten: als Haskell-Prädikat

```
rbInvariants :: RedBlackTree Integer -> Bool
rbInvariants tree =
    noRedRed tree &&
    blackSame tree &&
    allSorted tree
```



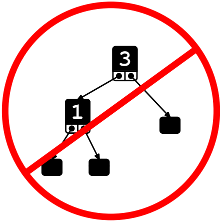
Implementierung:

```
noRedRed :: RedBlackTree t -> Bool
noRedRed = allSubtrees redRed
  where redRed (Node Red (Node Red a x b) y c) = False
        redRed (Node Red c y (Node Red a x b)) = False
        redRed x = True
```

Rot-Schwarz-Bäume

Invarianten: als Haskell-Prädikat

```
rbInvariants :: RedBlackTree Integer -> Bool
rbInvariants tree =
    noRedRed tree &&
    blackSame tree &&
    allSorted tree
```



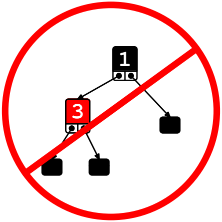
Implementierung:

```
blackSame :: RedBlackTree t -> Bool
blackSame tree = allSame (map countBlacks (colorPaths tree))
   where allSame (x:xs)      = all (==x) xs
         countBlacks         = length . (filter (==Black))

colorPaths :: RedBlackTree t -> [[Color]]
colorPaths tree = fold consAll [[]] tree
   where consAll c left y right = map (c:) (left++right)
```

Rot-Schwarz-Bäume

Invarianten: als Haskell-Prädikat

```
rbInvariants :: RedBlackTree Integer -> Bool
rbInvariants tree =
    noRedRed tree &&
    blackSame tree &&
    allSorted tree
```



Implementierung:

```
allSorted :: RedBlackTree Integer -> Bool
allSorted = allSubtrees sorted
   where sorted Leaf = True
         sorted (Node c left x right) = (allRB (<=x) left) &&
                                        (allRB (>=x) right)
```

Korrektheit der Implementierung

Korrektes Einfügen: Erhaltung der Invarianten

- Testdaten: Liste einzufügender Elemente

```
import Test.QuickCheck

fromList :: (Ord t) => [t] -> RedBlackTree t
fromList  = foldr insert Leaf

*Main> test (rbInvariants . fromList)
OK, passed 100 tests.
```

Korrektheit der Implementierung

Korrektes Einfügen: Funktionale Korrektheit

- Testdaten: Liste einzufügender Elemente

```
import Test.QuickCheck

rbInsertCorrect :: [Integer] -> Bool
rbInsertCorrect list =
 let
   tree = (fromList list)
 in
   all   ( `elemRB` tree) list &&
   allRB ( `elem`   list) tree

*Main> test rbInsertCorrect
OK, passed 100 tests.
```

Korrektheit der Implementierung

Korrekte Element-Suche: Funktionale Korrektheit

- Testdaten: Liste einzufügender Elemente, Liste abzufragender Elemente

```
import Test.QuickCheck

lookup :: (Ord t) => t -> RedBlackTree t -> Bool
lookup x Leaf  = False
lookup x (Node c left y right)
  | x==y = True
  | x<y  = lookup x left
  | x>y  = lookup x right

rbLookupCorrect :: RedBlackTree Integer -> [Integer] -> Bool
rbLookupCorrect tree tests = all sameOn tests
  where sameOn x = (lookup x tree) == (x `elemRB` tree)

*Main> test (rbLookupCorrect . fromList)
OK, passed 100 tests.
```