MapReduce als funktionales Skelett

Hauptseminar Multicore-Programmierung

Peter Lutz

Wintersemester 2010/2011
16. Dezember 2010

Motivation: Das Divide-and-Conquer-Skelett [FP2010, s03-18ff.]

```
dc :: (a->Bool)->(a->b)->(a->[a])->(a->[b]->b)->a->b
dc p b d c = dcprg
  where dcprg x = if p x
                  then b x
                  else c x (map dcprg (d x))
```

Motivation: Das Divide-and-Conquer-Skelett [FP2010, s03-18ff.]

```
dc :: (a->Bool)->(a->b)->(a->[a])->(a->[b]->b)->a->b
dc p b d c = dcprg
  where dcprg x = if p x
                  then b x
                  else c x (map dcprg (d x))
```

Instanziierung mit problemspezifischen Funktionen

- `p :: (a->Bool)` stellt fest, ob das Problem einfach ist.
- `b :: (a->b)` löst das Problem direkt.
- `d :: (a->[a])` teilt das Problem in eine Liste von Teilproblemen.
- `c :: (a->[b]->b)` verbindet die Teillösungen.

Motivation: Das Divide-and-Conquer-Skelett [FP2010, s03-18ff.]

```
dc :: (a->Bool)->(a->b)->(a->[a])->(a->[b]->b)->a->b
dc p b d c = dcprg
  where dcprg x = if p x
                  then b x
                  else c x (map dcprg (d x))
```

Instanziierung mit problemspezifischen Funktionen

- `p :: (a->Bool)` stellt fest, ob das Problem einfach ist.
- `b :: (a->b)` löst das Problem direkt.
- `d :: (a->[a])` teilt das Problem in eine Liste von Teilproblemen.
- `c :: (a->[b]->b)` verbindet die Teillösungen.

`dcprg :: (a->b)` ist das resultierende Programm.

Motivation: Anwendung von dc: funktionales „quicksort“

[FP2010, s03-21]

```
quicksort :: Ord a => [a] -> [a]
quicksort = dc p b d c
 where p xs     = length xs < 2
       b xs     = xs
       d (x:xs) = let (a,b) = partition (<x) xs
                  in [a,b]
       c (x:_) [a,b] = a ++ (x : b)
```

- `p :: ([a]->Bool)` stellt fest, ob das Problem einfach ist.
- `b :: ([a]->[a])` löst das Problem direkt.
- `d :: ([a]->[[a]])` teilt das Prob. in eine Liste von Teilprob.
- `c :: ([a]->[[a]]->[a])` verbindet die Teillösungen.

Ablauf

Weiterer Ablauf

Funktionale Skelette

Funktionales Skelett

Allgemeine Lösungsstrategie, Algorithmus für bestimmte Problemklasse, spezielle polymorphe Funktion höherer Ordnung:

nimmt sequentielle Funktion(en) als Parameter, die bestimmte Charakteristiken des konkretes Problems beschreiben

liefert (paralleles) Programm zur Lösung des konkreten Problems, wobei das Skelett die Berechnungsstrategie der (parallelen) Implementierung bestimmt

Beispiele

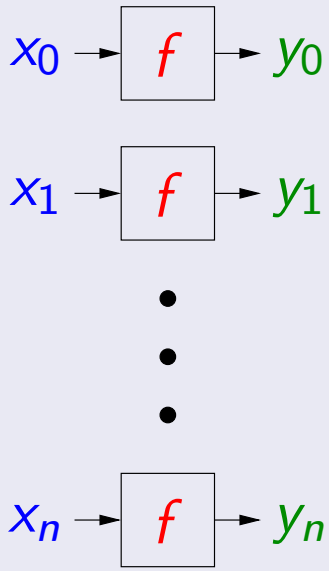
Divide-and-Conquer, Self-Service-Farm-Implementation of `map`, Replicated-Workers, MapReduce

Weiterer Ablauf

map [FP2010, s03]

```
map :: (a -> b) -> [a] -> [b]   -- Typ von map
map f []     = []               -- 1. Fall: leere Liste
map f (x:xs) = f x : map f xs   -- 2. Fall: nichtleere Liste,
                                --          Rekursion
```

map f $[x_0, \ldots, x_n] \rightsquigarrow [y_0, \ldots, y_n]$:

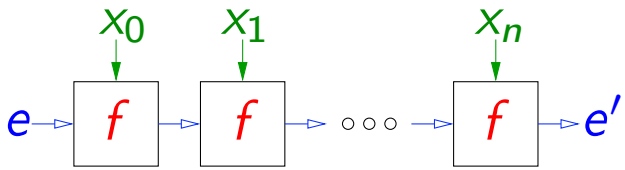


Beispiel

map ((*) 2) [1,2,3] $\rightsquigarrow$ [2,4,6]

foldl [FP2010, s03]

```
foldl :: (b -> a -> b) -> b -> [a] -> b   -- Typ von foldl
foldl f e []     = e                     -- 1.: leere Liste
foldl f e (x:xs) = foldl f (f e x) xs    -- 2.: nichtleere L.
```

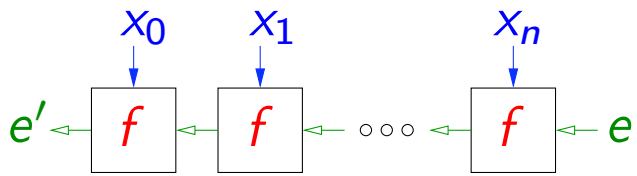


Beispiele

```
sum  [1,2,3,4] = foldl (+) 0 [1,2,3,4] = (((0+1)+2)+3)+4
prod [1,2,3,4] = foldl (*) 1 [1,2,3,4] = (((1*1)*2)*3)*4
```

foldr [FP2010, s03]

```
foldr :: (a -> b -> b) -> b -> [a] -> b   -- Typ von foldr
foldr f e []     = e                     -- 1.: leere Liste
foldr f e (x:xs) = f x (foldr f e xs)    -- 2.: nichtleere L.
```



Beispiel

```
and [a,b,c] = foldr (&&) True [a,b,c]
            = a && (b && (c && True))
```

Weiterer Ablauf

Vorbereitungen

Funktionskomposition mit Punktoperator

```
(g . f) x = g (f x)
```

Vorbereitungen

Funktionskomposition mit Punktoperator

```
(g . f) x = g (f x)
```

Der Datentyp Maybe

```
data Maybe v = Just v | Nothing
```

Vorbereitungen

Funktionskomposition mit Punktoperator

```
(g . f) x = g (f x)
```

Der Datentyp Maybe

```
data Maybe v = Just v | Nothing
```

Modul Data.Map: assoziatives Array (Dictionary) [Data.Map]

```
data Map k a    -- Ein Map von Schlüsseln k auf Werte a
```

Übersicht

```
mapReduce mapper reducer =
     reducePerKey    -- 3. Wende reducer auf jede Gruppe an
   . groupByKey      -- 2. Gruppiere Zwischendaten mittels Schlüssel
   . mapPerKey       -- 1. Wende mapper auf jedes Schlüssel/Wert-Paar an
 where
   reducePerKey :: Map k2 [v2] -> Map k2 v3
   groupByKey   :: [(k2,v2)]   -> Map k2 [v2]
   mapPerKey    :: Map k1 v1   -> [(k2,v2)]
```

Übersicht

```
mapReduce mapper reducer =
     reducePerKey    -- 3. Wende reducer auf jede Gruppe an
   . groupByKey      -- 2. Gruppiere Zwischendaten mittels Schlüssel
   . mapPerKey       -- 1. Wende mapper auf jedes Schlüssel/Wert-Paar an
 where
   reducePerKey :: Map k2 [v2] -> Map k2 v3
   groupByKey   :: [(k2,v2)]   -> Map k2 [v2]
   mapPerKey    :: Map k1 v1   -> [(k2,v2)]
```

Übersicht

```
mapReduce mapper reducer =
      reducePerKey    -- 3. Wende reducer auf jede Gruppe an
    . groupByKey      -- 2. Gruppiere Zwischendaten mittels Schlüssel
    . mapPerKey       -- 1. Wende mapper auf jedes Schlüssel/Wert-Paar an
  where
    reducePerKey :: Map k2 [v2] -> Map k2 v3
    groupByKey   :: [(k2,v2)]   -> Map k2 [v2]
    mapPerKey    :: Map k1 v1   -> [(k2,v2)]
```

Übersicht

```
mapReduce mapper reducer =
      reducePerKey    -- 3. Wende reducer auf jede Gruppe an
    . groupByKey      -- 2. Gruppiere Zwischendaten mittels Schlüssel
    . mapPerKey       -- 1. Wende mapper auf jedes Schlüssel/Wert-Paar an
  where
    reducePerKey :: Map k2 [v2] -> Map k2 v3
    groupByKey   :: [(k2,v2)]   -> Map k2 [v2]
    mapPerKey    :: Map k1 v1   -> [(k2,v2)]
```

Übersicht

```
mapReduce :: forall k1 k2 v1 v2 v3.
             (Ord k2)                       -- für Gruppieren
          => (k1 -> v1 -> [(k2,v2)])        -- die mapper-Funktion
          -> (k2 -> [v2] -> Maybe v3)       -- die reducer-Funktion
          -> Map k1 v1                      -- Eingabe-Schlüssel/Wert-Map
          -> Map k2 v3                      -- Ausgabe-Schlüssel/Wert-Map
```

```
mapReduce mapper reducer =
     reducePerKey    -- 3. Wende reducer auf jede Gruppe an
   . groupByKey      -- 2. Gruppiere Zwischendaten mittels Schlüssel
   . mapPerKey       -- 1. Wende mapper auf jedes Schlüssel/Wert-Paar an
 where
   reducePerKey :: Map k2 [v2] -> Map k2 v3
   groupByKey   :: [(k2,v2)]   -> Map k2 [v2]
   mapPerKey    :: Map k1 v1   -> [(k2,v2)]
```

Weiterer Ablauf

1. Phase: mapPerKey

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

```
fromList :: Ord k => [(k, a)] -> Map k a
toList   ::          Map k a  -> [(k, a)]

mapper :: k1 -> v1 -> [(k2,v2)]

uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f (a, b) = f a b

concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
    concat                      -- 3. Konkateniere Listen
  . map (uncurry mapper)        -- 2. Wende mapper auf jedes Element
                                --    der Liste von Paaren an
  . toList                      -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
mapPerKey (fromList
  [("t01","ihr seid alle individuen"),
   ("t02","wir sind alle individuen"),
   ("t03","ich nicht")])
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
    concat                      -- 3. Konkateniere Listen
  . map (uncurry mapper)        -- 2. Wende mapper auf jedes Element
                                --    der Liste von Paaren an
  . toList                      -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
mapPerKey (fromList
  [("t01","ihr seid alle individuen"),
   ("t02","wir sind alle individuen"),
   ("t03","ich nicht")])
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat . map (uncurry mapper) . toList (fromList
  [("t01","ihr seid alle individuen"),
   ("t02","wir sind alle individuen"),
   ("t03","ich nicht")])
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat . map (uncurry mapper) . toList (fromList
  [("t01","ihr seid alle individuen"),
   ("t02","wir sind alle individuen"),
   ("t03","ich nicht")])
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
    concat                      -- 3. Konkateniere Listen
  . map (uncurry mapper)        -- 2. Wende mapper auf jedes Element
                                --    der Liste von Paaren an
  . toList                      -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat . map (uncurry mapper)
  [("t01","ihr seid alle individuen"),
   ("t02","wir sind alle individuen"),
   ("t03","ich nicht")]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
    concat                      -- 3. Konkateniere Listen
  . map (uncurry mapper)        -- 2. Wende mapper auf jedes Element
                                --    der Liste von Paaren an
  . toList                      -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat . map (uncurry mapper)
  [("t01","ihr seid alle individuen"),
   ("t02","wir sind alle individuen"),
   ("t03","ich nicht")]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
    concat                      -- 3. Konkateniere Listen
  . map (uncurry mapper)        -- 2. Wende mapper auf jedes Element
                                --    der Liste von Paaren an
  . toList                      -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ (uncurry mapper) ("t01","ihr seid alle individuen"),
    (uncurry mapper) ("t02","wir sind alle individuen"),
    (uncurry mapper) ("t03","ich nicht") ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
    concat                      -- 3. Konkateniere Listen
  . map (uncurry mapper)        -- 2. Wende mapper auf jedes Element
                                --    der Liste von Paaren an
  . toList                      -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ (uncurry mapper) ("t01","ihr seid alle individuen"),
    (uncurry mapper) ("t02","wir sind alle individuen"),
    (uncurry mapper) ("t03","ich nicht") ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ (uncurry mapper) ("t01","ihr seid alle individuen"), ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ (uncurry mapper) ("t01","ihr seid alle individuen"), ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ mapper "t01" "ihr seid alle individuen", ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ mapper "t01" "ihr seid alle individuen", ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
    concat                      -- 3. Konkateniere Listen
  . map (uncurry mapper)        -- 2. Wende mapper auf jedes Element
                                --    der Liste von Paaren an
  . toList                      -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ const (map (flip (,) 1) . words) "t01" "ihr seid alle individuen",
    ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
    concat                      -- 3. Konkateniere Listen
  . map (uncurry mapper)        -- 2. Wende mapper auf jedes Element
                                --    der Liste von Paaren an
  . toList                      -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ const (map (flip (,) 1) . words) "t01" "ihr seid alle individuen",
    ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ (map (flip (,) 1) . words) "ihr seid alle individuen", ...]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ (map (flip (,) 1) . words) "ihr seid alle individuen", ...]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ map (flip (,) 1) ["ihr", "seid", "alle", "individuen"], ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ map (flip (,) 1) ["ihr", "seid", "alle", "individuen"], ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ [ flip (,) 1 "ihr",
      flip (,) 1 "seid",
      flip (,) 1 "alle",
      flip (,) 1 "individuen" ], ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
    concat                      -- 3. Konkateniere Listen
  . map (uncurry mapper)        -- 2. Wende mapper auf jedes Element
                                --    der Liste von Paaren an
  . toList                      -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ [ flip (,) 1 "ihr",
      flip (,) 1 "seid",
      flip (,) 1 "alle",
      flip (,) 1 "individuen" ], ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ [ flip (,) 1 "ihr", ... ], ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ [ flip (,) 1 "ihr", ... ], ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ [ (,) "ihr" 1, ... ], ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ [ (,) "ihr" 1, ... ], ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ [ ("ihr", 1), ... ], ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
concat
  [ [ ("ihr", 1), ... ], ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
    concat                    -- 3. Konkateniere Listen
  . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                              --    der Liste von Paaren an
  . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
[ ("ihr", 1), ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
    concat                    -- 3. Konkateniere Listen
  . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                              --    der Liste von Paaren an
  . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
[ ("ihr", 1), ... ]
```

1. Phase: mapPerKey: Beispiel

```
mapPerKey :: Map k1 v1 -> [(k2,v2)]
mapPerKey =
   concat                    -- 3. Konkateniere Listen
 . map (uncurry mapper)      -- 2. Wende mapper auf jedes Element
                             --    der Liste von Paaren an
 . toList                    -- 1. Verwandle Map in Liste
```

Wörter zählen

```
mapper = const (map (flip (,) 1) . words)
```

```
[ ("ihr", 1), ("seid", 1), ("alle", 1), ("individuen", 1),
  ("wir", 1), ("sind", 1), ("alle", 1), ("individuen", 1),
  ("ich", 1), ("nicht", 1) ]
```

Weiterer Ablauf

2. Phase: groupPerKey

```
groupByKey :: [(k2,v2)] -> Map k2 [v2]
groupByKey = foldl insert empty
```

2. Phase: groupPerKey

```
groupByKey :: [(k2,v2)] -> Map k2 [v2]
groupByKey = foldl insert empty
```

```
  where
    insert dict (k2,v2) = insertWith (++) k2 [v2] dict
```

2. Phase: groupPerKey

```
groupByKey :: [(k2,v2)] -> Map k2 [v2]
groupByKey = foldl insert empty
```

```
  where
    insert dict (k2,v2) = insertWith (++) k2 [v2] dict
```

```
empty :: Map k a == fromList []    -- leere Map

insertWith :: Ord k => (a -> a -> a) -> k -> a -> Map k a -> Map k a

-- fügt das Paar (Schlüssel, Wert) in Map ein,
--    falls Schlüssel nicht existiert

-- ersetzt das Paar (Schlüssel, alter_Wert) durch
--    (Schlüssel, f neuer_Wert alter_Wert),
--    falls Schlüssel existiert
```

2. Phase: groupPerKey

```
groupByKey :: [(k2,v2)] -> Map k2 [v2]
groupByKey = foldl insert empty
```

```
  where
    insert dict (k2,v2) = insertWith (++) k2 [v2] dict
```

Beispiel

```
[ ("ihr", 1), ("seid", 1), ("alle", 1), ("individuen", 1),
  ("wir", 1), ("sind", 1), ("alle", 1), ("individuen", 1),
  ("ich", 1), ("nicht", 1) ]

fromList [ ("alle", [1, 1]), ("ich", [1]), ("ihr", [1]),
           ("individuen", [1, 1]), ("nicht", [1]), ("seid", [1]),
           ("sind", [1]), ("wir", [1]) ]
```

Weiterer Ablauf

3. Phase: reducePerKey

```
reducePerKey :: Map k2 [v2] -> Map k2 v3
reducePerKey =
    mapWithKey unJust          -- 3. Verwandle Typ um Maybe zu entfernen
  . filterWithKey isJust       -- 2. Entferne Einträge mit Wert Nothing
  . mapWithKey reducer         -- 1. Wende reducer pro Schlüssel an
```

3. Phase: reducePerKey

```
reducePerKey :: Map k2 [v2] -> Map k2 v3
reducePerKey =
    mapWithKey unJust        -- 3. Verwandle Typ um Maybe zu entfernen
  . filterWithKey isJust     -- 2. Entferne Einträge mit Wert Nothing
  . mapWithKey reducer       -- 1. Wende reducer pro Schlüssel an
```

```
  where
    isJust k (Just v)   = True    -- Behalte  derartige Einträge
    isJust k Nothing    = False   -- Entferne derartige Einträge
    unJust k (Just v)   = v       -- Verwandle optionalen in
                                  --   nicht-optionalen Typ
```

3. Phase: reducePerKey

```
reducePerKey :: Map k2 [v2] -> Map k2 v3
reducePerKey =
    mapWithKey unJust          -- 3. Verwandle Typ um Maybe zu entfernen
  . filterWithKey isJust       -- 2. Entferne Einträge mit Wert Nothing
  . mapWithKey reducer         -- 1. Wende reducer pro Schlüssel an
```

```
  where
    isJust k (Just v)   = True    -- Behalte  derartige Einträge
    isJust k Nothing    = False   -- Entferne derartige Einträge
    unJust k (Just v)   = v       -- Verwandle optionalen in
                                  --   nicht-optionalen Typ
```

```
reducer        :: k2 -> [v2] -> Maybe v3
mapWithKey     ::               (k -> a -> b) -> Map k a -> Map k b
filterWithKey :: Ord k => (k -> a -> Bool) -> Map k a -> Map k a
```

3. Phase: reducePerKey

```
reducePerKey :: Map k2 [v2] -> Map k2 v3
reducePerKey =
    mapWithKey unJust          -- 3. Verwandle Typ um Maybe zu entfernen
  . filterWithKey isJust       -- 2. Entferne Einträge mit Wert Nothing
  . mapWithKey reducer         -- 1. Wende reducer pro Schlüssel an
```

```
reducer = const (Just . sum)  -- Berechne die Summe aller Zählungen
```

Beispiel

```
fromList [ ("alle", [1, 1]), ("ich", [1]), ("ihr", [1]),
           ("individuen", [1, 1]), ("nicht", [1]), ("seid", [1]),
           ("sind", [1]), ("wir", [1]) ]

fromList [ ("alle", 2), ("ich", 1), ("ihr", 1), ("individuen", 2),
           ("nicht", 1), ("seid", 1), ("sind", 1), ("wir", 1)]
```

Weiterer Ablauf

combiner

Motivation

- beschränkte Bandbreite des Netzwerkes
- deshalb Beachtung der Datenlokalität durch:
  - Daten für Map-Tasks möglichst im lokalen Speicher
  - Reduzierung der Zwischendaten, die von Map-Tasks zu Reduce-Tasks geschickt werden

combiner

Motivation

- beschränkte Bandbreite des Netzwerkes
- deshalb Beachtung der Datenlokalität durch:
  - Daten für Map-Tasks möglichst im lokalen Speicher
  - Reduzierung der Zwischendaten, die von Map-Tasks zu Reduce-Tasks geschickt werden

```
combiner :: k2 -> [v2] -> Maybe v3
```

- wird nach `map` auf Map-Task ausgeführt, um lokale Zwischendaten zu reduzieren

Weiterer Ablauf

Möglichkeiten für Parallelität im MapReduce-Modell (1)

Paralleler map über die Eingabedaten

- map auf jedes einzelne Element $\longrightarrow$ totale Datenparallelität
- Voraussetzung: Reihenfolge der Auswertung spielt keine Rolle

Möglichkeiten für Parallelität im MapReduce-Modell (1)

Paralleler map über die Eingabedaten

- map auf jedes einzelne Element → totale Datenparallelität
- Voraussetzung: Reihenfolge der Auswertung spielt keine Rolle

Paralleles Gruppieren der Zwischendaten

- Sortierproblem → parallele Sortiermodelle
- kann mit map-Phase verbunden werden

Möglichkeiten für Parallelität im MapReduce-Modell (2)

Paralleler map über Gruppen

Reduktion einzeln für jede Gruppe $\longrightarrow$ totale Datenparallelität

Möglichkeiten für Parallelität im MapReduce-Modell (2)

Paralleler map über Gruppen

Reduktion einzeln für jede Gruppe $\longrightarrow$ totale Datenparallelität

Parallele Reduktion pro Gruppe

- wenn Reduktionsfunktion eine *eigentliche Reduktion* ist, kann jede Anwendung der Reduktion massiv parallelisiert werden
- Berechnung von Unter-Reduktionen in einer Baum-Struktur
- assoziative Operation auf die Knoten angewendet

eigentliche Reduktion

- Eingabefunktion der Reduktion frei von Seiteneffekten
- assoziative, zweistellige Funktion
- Einheit entspricht dem Initialwert

Weiterer Ablauf

Eden

Eden

- erweitert Haskell um Konstrukte zur Kontrolle der parallelen Auswertung von Prozessen
- kümmert sich um Kommunikation und Synchronisation – transparent für Benutzer
- ist als parallele funktionale Sprache geeignet, um anspruchsvolle Skelette zu entwickeln

Komposition von `map` und `fold`

```
mapFoldL :: (a -> b) -> (c -> b -> c) -> c -> [a] -> c
mapFoldL map reduce n list = foldl reduce n (map map list)
```

```
mapFoldR :: (a -> b) -> (b -> c -> c) -> c -> [a] -> c
mapFoldR map reduce n list = foldr reduce n (map map list)
```

- für parallele Implementierungen müssen Typen `b` und `c` übereinstimmen
- Assoziativität muss in den getrennten Unter-Reduktionen gelten.
- `n` sollte neutrales Element von *reduce* sein.
- Für Umordnung Kommutativität von *reduce* nötig.

Parallele MapReduce-Implementierung in Eden

```
parmapFoldL :: (Trans a, Trans b) =>
               Int ->                    -- Anzahl der Prozesse
               (a -> b) ->               -- map auf die Eingabe
               (b -> b -> b) ->          -- reduce (kommutativ)
               b ->                      -- neutrales Element der Reduktion
               [a] -> b

parmapFoldL np map reduce neutral list =
  foldl' reduce neutral subRs
    where sublists    = unshuffle np list
          subFoldProc = process (foldl' reduce neutral . (map map))
          subRs       = spawn (replicate np subFoldProc) sublists

unshuffle :: Int -> [a] -> [[a]]        -- verteilt Eingabestrom per ...
unshuffle n list = ...                  -- ... round-robin auf np Ströme

spawn :: [Process a b] -> [a] -> [b]    -- erzeugt Menge von Prozessen ...
spawn ps inputs = ...                   -- ... mit zugehörigen Eingaben
```

Weiterer Ablauf

Alternative Skelette

Ungeeignete Probleme

- `MapReduce` für Probleme nicht geeignet, die iterativ in festgelegten Schritten arbeiten; wenn Datenvektoren in jedem Schritt ausgetauscht werden
- Kosten für Erstellung einer neuen `MapReduce`-Instanz für jede Iteration zu hoch
- Bsp.: *k-means*
- Iterator-Skelett hier besser geeignet

Mehrmalige Ausführung

Einige Probleme lassen sich nicht in einer `MapReduce`-Ausführung lösen, aber durch mehrmaliges Ausführen, wobei Ausgabe eines Durchlaufs als Eingabe des nächsten dient.

Fazit

- Funktionale Skelette neben Design Patterns und Refactorings weitere Säule des gehobenen Software Engineerings
- noch zu wenig Beachtung, nach `MapReduce` aber Aufmerksamkeit gestiegen
- Wahl des richtigen Skeletts erfordert eingehende Analyse des Problems, sonst: ein Skelett als goldener Hammer
- interessantes und spannendes Forschungsfeld

Wann `foldl` und wann `foldr`? [FP2010, s03]

`foldl`

Für strikte Operatoren wie + bevorzugt man `foldl`, weil

1. die Zahlen in der Liste nicht extra gespeichert werden müssen und
2. sowieso die ganze Liste durchgegangen werden muss.

`foldr`

Für im zweiten Argument nicht-strikte Operatoren wie && bevorzugt man `foldr`, weil

1. `foldl` die ganze Liste durchlaufen würde, aber
2. das Ergebnis bereits früher feststehen kann.

Literatur

Lämmel, Ralf:
Google's MapReduce programming model – Revisited.
In: *Science of Computer Programming* 70 (2008), Nr. 1, 1 - 30.
http://dx.doi.org/DOI:10.1016/j.scico.2007.07.001.

Herrmann, C.; Griebl, M.; Lengauer, C.; Größlinger, A.:
Skript: Funktionale Programmierung, Sommersemester 2010.
Universität Passau, Lehrstuhl für Programmierung.

Haskell.org
Data.Map: An efficient implementation of maps from keys to values (dictionaries).
http://www.haskell.org/ghc/docs/latest/html/libraries/containers/Data-Map.html.