`MapReduce` als funktionales Skelett

Peter Lutz

16. Dezember 2010

Hauptseminar Multicore-Programmierung
Wintersemester 2010/2011

Abstract

Diese Arbeit ordnet Googles `MapReduce`-Programmiermodel als funktionales Skelett ein. Dazu werden die aus der funktionalen Programmierung bekannten Kombinatoren `map` und `reduce` betrachtet und darauf aufbauend eine erste Implementierung von `MapReduce` in *Haskell* vorgestellt. Für diese Implementierung werden dann Möglichkeiten der Parallelisierung aufgezeigt und die Implementierung entsprechend erweitert. Danach wird eine zweite, parallele Implementierung in der parallelen Programmiersprache *Eden* erläutert.

Inhaltsverzeichnis

1 Einleitung

In ihrem Artikel „`MapReduce`: *Simplified Data Processing on Large Clusters*“ [DG04][1] stellten im Jahr 2004 Jeffrey Dean und Sanjay Ghemawat ein Programmiermodell (und eine zugehörige Implementierung) namens `MapReduce` vor. Dieses Programmiermodell ist in der Lage, große Datenmengen parallel zu verarbeiten und zu erzeugen. Der Benutzer spezifiziert durch Angabe einer `map`- und einer `reduce`-Funktion die konkrete Berechnung. Die Implementierung bewerkstelligt eine automatische Parallelisierung der Berechnung und kümmert sich um die Verteilung der Daten auf die einzelnen Arbeitsknoten, der Zuweisung der Arbeitspakete an die Arbeitsknoten, die Behandlung von Rechnerausfällen und die Kommunikation zwischen den Rechnern.

Die Autoren beschreiben ein mögliches Implementierungsmodell, das von großen – durch ein Netzwerk verbundenen – Clustern aus handelsüblichen Rechnern mit lokalem Speicher ausgeht. Auch wenn das im Folgenden beschriebene Modell sehr einschränkend erscheinen mag, so bietet es doch einen guten Ansatz für viele Probleme, die bei der Verarbeitung großer Datenmengen auftreten. Außerdem können durch die Zerlegung eines Problems in mehrere `MapReduce`-Berechnungen oder indem man Teilprobleme durch andere, weniger restriktive Modelle berechnen lässt, mögliche Einschränkungen beseitigt werden. [Läm08]

Dieses Programmiermodell stellt eine Abstraktion dar, die als funktionales Skelett bekannt ist. Eine Definition von funktionalen Skeletten wird im nächsten Abschnitt gegeben. Diese Arbeit ordnet das `MapReduce`-Programmiermodell in die Theorie der funktionalen Skelette ein.

Diese Arbeit setzt grundlegende Kenntnisse der funktionalen Programmiersprache Haskell voraus, insbesondere einiger oft verwendeter Funktionen der Haskell-Prelude. Allerdings werden einige – für das Verständnis zentrale – Funktionen, vor allem Kombinatoren, genauer erklärt.

Abschnitt 2 liefert eine Definition von funktionalen Skeletten. Abschnitt 3 liefert eine Einordnung des `MapReduce`-Ansatzes in die Theorie der funktionalen Skelette. Dazu werden die aus vielen funktionalen Programmiersprachen bekannten Kombinatoren `map` und `reduce` erläutert und eine Implementierung des `MapReduce`-Modells in Form eines grundlegenden Programm-Skeletts in der funktionalen Programmiersprache Haskell erarbeitet. Gelegenheiten für parallele Ausführung der Programmteile werden aufgezeigt und die Implementierung entsprechend erweitert. In Abschnitt 4 wird eine zweite, parallele Implementierung in der parallelen funktionalen Programmiersprache Eden erläutert.

[1] Überarbeitete Version 2008: [DG08]

2 Was ist ein funktionales Skelett?

Algorithmische Skelette wurden 1989 von Murray Cole in seiner Doktorarbeit als spezielle Funktionen höherer Ordnung beschrieben. Eine Funktion höherer Ordnung ist ein Programm- oder ein Prozedur-Template, das die Gesamtstruktur der Berechnung festlegt, dabei aber „Lücken“ für die Definition von problemspezifischen Prozeduren und Deklarationen freilässt. Im funktionalen Paradigma sind dies Funktionen, die Funktionen als Parameter erwarten oder Funktionen zurückliefern. Der Programmierer legt nun Funktionen fest, auf die die Funktion höherer Ordnung angewendet wird, um ein problemspezifisches Gesamtprogramm zu erzeugen. Dieses Programm löst das konkrete Problem. [Col91]

Ein algorithmisches (oder funktionales) Skelett beschreibt also auf abstrakte Weise die Struktur eines Algorithmus, trennt dabei aber die Spezifikation des Lösungsansatzes des konkreten Problems als eine oder mehrere zu übergebende Funktionen ab. Das Skelett beinhaltet Potential für eine inhärente Parallelisierung im Algorithmus, das allerdings in der Implementierung des Skeletts versteckt ist. Die zu übergebende Funktion bliebt sequentiell. [BDL09]

Cole [Col04] ist der Meinung, dass die Programmierung mit Skeletten es anbietet, dass von generischen Mustern der Berechnung und Kommunikation (wie zum Beispiel dem Pipeline-Paradigma) abstrahiert und sie als Werkzeugkasten für den Programmierer angeboten werden können. Diese Abstraktion würde Spezifikationen beinhalten, die über die architektonischen Variationen erhaben sind, aber auch Implementierungen, die diese Variationen erkennen können, um die Performanz zu steigern.

> „Auf diese Weise verspricht [das Programmieren mit Skeletten], viele der üblichen Probleme innerhalb des parallelen Software-Engineering-Prozesses anzusprechen:
>
> - es wird das Programmieren *vereinfachen*, indem der Grad der Abstraktion erhöht wird;
> - es wird die *Portabilität* und die *Wiederbenutzung* fördern, indem der Programmierer von der Verantwortung für die detailierte Realisierung der zugrundeliegenden Muster entbunden wird;
> - es wird die *Performanz* verbessern, indem Zugang zu sorgfältig optimierten, architekturspezifischen Implementierungen der Muster gewährt wird;
> - es wird Möglichkeiten für statische und dynamische *Optimierungen* bieten, indem explizit Informationen über die algorithmische Struktur dokumentiert werden (z.B. über die Verteilung und die Abhängigkeiten), die oftmals aus gleichwertigen, aber unstrukturierten Programmen unmöglich entnommen werden können.“
>
> [Col04, Übersetzung: Peter Lutz]

3 Warum ist `MapReduce` ein funktionales Skelett?

Googles `MapReduce`-Modell wurde hauptsächlich von zwei Funktionen – sogenannten Kombinatoren – motiviert, die sich in vielen funktionalen Sprachen (wie zum Beispiel Lisp) finden: `map` und `reduce`. [DG04] Diese Kombinatoren sind Funktionen höherer Ordnung, da sie Funktionen als Parameter nehmen.

3.1 Das `MapReduce`-Programmiermodell

Googles `MapReduce`-Programmiermodell wird von den Autoren folgendermaßen beschrieben:

> „Die Berechnung nimmt eine Menge von Eingabe-Schlüssel/Wert-Paaren und liefert eine Menge von Ausgabe-Schlüssel/Wert-Paaren. Der Benutzer der `MapReduce`-Bibliothek spezifiziert die Berechnung durch zwei Funktionen: *map* und *reduce*.
>
> *map*, das vom Benutzer geschrieben wird, nimmt ein Eingabe-Paar und liefert eine Menge von Zwischen-Schlüssel/Wert-Paaren. Die `MapReduce`-Bibliothek gruppiert alle Zwischenwerte, die zum selben Zwischenschlüssel I gehören und leitet diese an die Reduktionsfunktion weiter.
>
> Die Reduktionsfunktion, die auch vom Benutzer geschrieben wird, nimmt einen Zwischenschlüssel I und eine Menge von Werten zu diesem Schlüssel entgegen. Sie fasst diese Werte zusammen, um eine möglicherweise kleinere Menge von Werten zu bilden. Pro Aufruf der Reduktion wird typischerweise kein oder nur ein Ausgabewert erzeugt. Die Zwischenwerte werden per Iterator an die Reduktionsfunktion des Benutzers weitergegeben. Dies erlaubt es uns, mit Listen von Werten umzugehen, die zu groß für den Speicher sind.“ [DG04, Übersetzung: Peter Lutz]

Das Programmiermodell basiert also auf folgenden einfachen Konzepten:

1. **Iteration** über die Eingabe
2. **Berechnung** der Schlüssel/Wert-Paare aus jedem Teil der Eingabe
3. **Gruppieren** aller Zwischenwerte mittels des Schlüssels
4. **Iteration** über die entstandenen Gruppen
5. **Reduktion** jeder einzelnen Gruppe

Als Beispiel für eine Spezifikation der zwei Funktionen *map* und *reduce* wird von den Autoren folgender Pseudocode angeführt, der das Vorkommen eines jeden Wortes in einer großen Dokumentensammlung zählt:

```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
 EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
 result += ParseInt(v);
Emit(AsString(result));
```

> „Die `map`-Funktion gibt jedes Wort zusammen mit seiner Häufigkeit aus (in diesem einfachen Beispiel: `1`). Die Reduktionsfunktion summiert alle Zählungen, die für ein bestimmtes Wort ausgegeben wurden.“ [DG04, Übersetzung: Peter Lutz]

Bei der Benennung der Funktionen sind in dieser Arbeit drei Ebenen zu unterscheiden:

- die Funktionen höherer Ordnung (sogenannte Kombinatoren) für das Mappen und die Reduktion: `map` und `reduce`,
- die Funktionen als Argumente der Kombinatoren: *map* und *reduce* und
- die eigentliche Anwendung der Kombinatoren auf die Argumente.

Diese drei Ebenen werden in dem ursprünglichen `MapReduce`-Artikel an manchen Stellen durcheinandergebracht.

Rekursionsschemata wie *map* und *reduce* erlauben mächtige Formen der Dekomposition und der Wiederverwendung. Sie drängen geradezu zu paralleler Ausführung, wenn die problemspezifischen Bestandteile frei von Seiteneffekten sind und gewisse algebraische Eigenschaften erfüllen. Da sich die Autoren des `MapReduce`-Artikels bei der Entwicklung von `MapReduce` von den Funktionen `map` und `reduce` aus Lisp inspirieren ließen, werden diese Funktionen an dieser Stelle erläutert:

3.2 Lisps `map` und `reduce`

In der Programmiersprache Lisp gibt es einen Kombinator namens `map`, der eine Funktion und mindestens eine oder mehrere Sequenzen als Eingabe nimmt. Dabei muss die Eingabefunktion soviele Argumente entgegennehmen, wie Sequenzen gegeben sind. Der Kombinator liefert eine Sequenz zurück, die so lang ist wie die kürzeste der Eingabesequenzen, und deren j-tes Element das Ergebnis der Anwendung der Eingabefunktion auf die Folge der j-ten Elemente der Eingabesequenzen ist. [Ste90, 14.2]

In Haskell entspricht der Fall einer Eingabesequenz dem Kombinator `map` und der Fall zweier Eingabesequenzen dem Kombinator `zipWith`.

Desweiteren gibt es in Lisp den Kombinator `reduce`, der alle Elemente einer Sequenz mittels einer zweistelligen Operation auf einen Wert reduziert. Es ist möglich, nur eine Teilsequenz zu reduzieren. Die Reduktion erfolgt standardmäßig linksassoziativ, kann aber durch Angabe eines Parameters auch rechtsassoziativ durchgeführt werden. Optional

kann ein Initialwert angegeben werden, der im linksassoziativen Fall vor der Sequenz und im rechtsassoziativen Fall hinter der Sequenz platziert und in die Reduktion einbezogen wird. Ist die Sequenz einelementig und kein Initialwert angegeben, so wird das Element der Sequenz zurückgegeben. Ist die Sequenz leer und ein Initialwert angegeben, so wird der Initialwert zurückgegeben. In beiden Fällen wird die Eingabefunktion nicht aufgerufen. Ist die Sequenz leer und kein Initialwert angegeben, so wird die Eingabefunktion ohne Argumente aufgerufen und die Reduktion gibt das Ergebnis der Eingabefunktion zurück. [Ste90, 14.2]

Eigentliche Reduktion Im Allgemeinen wird angenommen, dass die Eingabefunktion der Reduktion frei von Seiteneffekten und eine assoziative, zweistellige Funktion ist, deren Einheit dem Initialwert entspricht. In diesem Fall spricht man von einer *eigentlichen Reduktion*.

3.3 Haskells Kombinatoren

Haskells Prelude definiert ähnliche Kombinatoren wie `map`, der eine einzige Liste entgegennimmt (im Gegensatz zu `map` in Lisp). Die linksassoziative Reduktion von Lisp wird in Haskell durch den Kombinator `foldl` bewerkstelligt, wobei der Typ von `foldl` genereller ist, als dies für eine Reduktion notwendig ist. Die rechtsassoziative Reduktion wird in Haskell durch den Kombinator `foldr` ausgedrückt.

Haskells `map`

```
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
```

Der Kombinator `map` nimmt zwei Argumente: eine Funktion vom Typ `a -> b` und eine Liste vom Typ `[a]`. Der Kombinator liefert als Ergebnis eine Liste vom Typ `[b]`. Die Typvariablen `a` und `b` der übergebenen Funktion entsprechen den Elementtypen der Eingabe- und Ausgabelisten. Eine Anwendung des Kombinators auf die leere Liste liefert die leere Liste zurück. Bei Anwendung des Kombinators auf eine nichtleere Liste wird das Ergebnis der Anwendung der Funktion auf das erste Element der Liste vorne an das Ergebnis der rekursiven Anwendung des Kombinators auf die Liste der restlichen Elemente angefügt.

Insgesamt erzeugt der Kombinator also ein Liste von Elementen, wobei jedes Element das Ergebnis der Anwendung der Funktion auf das ursprüngliche Element der Liste ist.

Beispiel:

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

Haskells `foldl`

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

Der Kombinator `foldl` nimmt drei Argumente: eine zweistellige Funktion vom Typ `b -> a -> b`, einen Initialwert vom Typ `b` und eine Liste vom Typ `[a]`. Der Kombinator liefert als Ergebnis einen Wert vom Typ `b`. Die Anwendung des Kombinators auf die leere Liste liefert den Initialwert zurück. Die Anwendung des Kombinators auf eine nichtleere Liste liefert zunächst das Ergebnis der Anwendung der Funktion (und des Initialwertes) auf das erste Element der Liste. Dieses Ergebnis wird anschließend als „Initialwert“ des nächsten rekursiven Schrittes verwendet.

Insgesamt reduziert der Kombinator die Liste linksassoziativ – ausgehend vom Initialwert – auf einen Ergebniswert.

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
```

Die Kombinatoren `map` und `foldl` können beide durch die rechtsassoziative Reduktion `foldr` ausgedrückt werden. Deshalb kann man `foldr` als das fundamentale Rekursionsschema für die Listentraversierung ansehen. Die Erläuterung wichtiger Kombinatoren von Haskell wird an dieser Stelle mit `foldr` abgeschlossen:

Haskells `foldr`

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

Der Kombinator `foldr` verhält sich im Wesentlichen wie der Kombinator `foldl`, allerdings wird die Liste von rechts nach links ausgewertet. Dadurch ändert sich auch der Typ der zu übergebenden Funktion zu `a -> b -> b`, da der Initialwert von Typ `b` nun von rechts an das letzte Element herangeführt wird, d.h. als zweites Argument in die Anwendung der Funktion auf das letzte Element der Liste eingeht.

Beispiel:

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

Reduktion durch Typspezialisierung Wir können `foldl` wie folgt einschränken:

```
reduce :: (a -> a -> a) -> a -> [a] -> a
reduce = foldl
```

Dies wird normalerweise „Reduktion durch Typspezialisierung“ genannt.

3.4 `MapReduce`s *map* und *reduce*

Wie lassen sich nun die Funktionen, die Googles `MapReduce`-Ansatz als Eingabe erwartet, in die Welt der Kombinatoren einordnen? Es wird sich im Laufe der Betrachtungen herausstellen, dass die Funktion *map*, die `MapReduce` als Eingabe erwartet, kein Kombinator

in der funktionalen Programmierung ist, sondern die *Funktion*, die der `map`-*Kombinator als Eingabe* erwartet. Die von `MapReduce` als Eingabe erwartete Funktion *reduce* dagegen ist typischerweise eine Anwendung des `reduce`-Kombinators und dient als Eingabe für den `map`-Kombinator. In der zu entwickelnden Haskell-Spezifikation von `MapReduce` wendet der `map`-Kombinator die Funktionen *map* und *reduce* auf sämtliche Eingabe- bzw. Zwischendaten an.

reduce Die Funktion *reduce* führt typischerweise eine Reduktion durch, d.h. sie ist eine Anwendung des Standard-`reduce`-Kombinators. Allerdings wird dem Programmierer nicht nahegelegt, die Bestandteile der Reduktion, nämlich die assoziative Operation und deren Einheit, zu identifiziert. Der Typ von *reduce* weicht auch von dem zu erwartenden Typ einer Reduktion ab, vermutlich um die Flexibilität zu erhöhen. Im `MapReduce`-Artikel werden nämlich Beispiele angeführt, in denen *reduce* Sortierung und Filterung durchführt.

map Im Beispiel des invertierten Index aus dem `MapReduce`-Artikel produziert *map* Listen anstatt sie zu konsumieren. Dies kann als eine Art von *unfolding* angesehen werden.

3.5 Zerlegung von MapReduce in Phasen

Im Folgenden wird eine ausführbare Spezifikation in Haskell vorgestellt, die eine Abstraktion der `MapReduce`-Berechnung darstellt. Die eigentliche Berechnung, die als Eingabe die Funktionen *mapper* und *reducer* nimmt, wird dabei zunächst in drei Phasen unterteilt, die nacheinander ausgeführt werden. Die Funktion *mapper* wird auf die Eingabe-Schlüssel/Wert-Paare gemappt und *reducer* auf geeignet gruppierte Zwischendaten. Dabei findet die Gruppierung zwischen der Map- und der Reduktionsphase statt, wie von den Autoren beschrieben:

> „Die `MapReduce`-Bibliothek gruppiert alle Zwischenwerte, die zum selben Zwischenschlüssel I gehören und leitet diese an die Reduktionsfunktion weiter.“ [DG04, Übersetzung: Peter Lutz]

Der Typ der Spezifikation ergibt sich zum einen aus den Typen der Funktionen *map* und *reduce*, die im `MapReduce`-Artikel folgendermaßen angegeben sind: [DG04]

```
map    (k1, v1)       -> list(k2, v2)
reduce (k2, list(v2)) -> list(v2)
```

Zum anderen wird der Typ einer `MapReduce`-Berechnung informell gegeben als:

> „Die Berechnung nimmt eine Menge von Eingabe-Schlüssel/Wert-Paaren und erzeugt eine Menge von Ausgabe-Schlüssel/Wert-Paaren“ [DG04, Übersetzung: Peter Lutz]

Das Beispiel, in dem die Häufigkeit der Wörter in Dokumenten bestimmt wird, lässt vermuten, dass hier statt Mengen eigentlich Listen gemeint sind, da mehrere Paare mit dem gleichen Wort vorkommen können und diese auch gezählt werden sollen. In Haskell ergibt diese Überlegung folgendes:

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

Der Typ für *reduce* wird so verändert, dass „typischerweise [...] null oder ein Ausgabewert pro Aufruf von *reduce* erzeugt [wird].“ [DG04, Übersetzung: Peter Lutz]. Dies wird durch den Datentyp `Maybe` ausgedrückt:

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

Ein Schlüssel mit dem Reduktionsergebnis `Nothing` sollte allerdings nicht zur Ergebnisliste der Schlüssel/Wert-Paare beitragen. Deshalb wird `Maybe` im Ergebnistyp von `MapReduce` weggelassen. Um den in einigen `MapReduce`-Szenarien vorhandenen Sortierungs- und Filterungsfunktionalitäten für die Reduktion Rechnung zu tragen, werden zwei verschiedene Typen angenommen: `v2` für Zwischenwerte und `v3` für Ausgabewerte. Die Ein- und Ausgabe-Schlüssel/Wert-Paare sind weder Mengen noch Listen, sondern ein assoziatives Array, da bei der Eingabe für den gleichen Schlüssel keine duplizierten Werte auftreten dürfen und bei der Ausgabe der gleiche Schlüssel nicht zweimal auftreten darf. Deshalb kommt hier ein Dictionary-Typ zum Einsatz, den der abstrakte Datentyp `Data.Map.Map` liefert (siehe nächster Abschnitt).

Insgesamt ergibt sich folgendes:

```
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
```

Die einzelnen Phasen werden mit Haskells Punktoperator zusammengekoppelt, wobei die am weitesten rechts stehende Funktion zuerst angewendet wird und dann von rechts nach links weiterverfahren wird: `(g . f) x = g (f x)`. Die Ausgabedaten einer Funktion dienen als Eingabedaten für die nächste Funktion.

`Data.Map` Um mit Schlüssel/Wert-Paaren effizient arbeiten zu können, wird der abstrakte Datentyp `Data.Map.Map` aus dem Modul `Data.Map` [has] verwendet, der ein assoziatives Array (Map, Dictionary) implementiert. Folgende Funktionen des Datentyps werden gebraucht:

- `toList :: Map k a -> [(k, a)]`
  exportiert eine Map als Liste von Paaren
- `fromList :: Ord k => [(k, a)] -> Map k a`
  erzeugt eine Map aus einer Liste von Schlüssel/Wert-Paaren

- `empty :: Map k a`
  die leere Map
- `singleton :: k -> a -> Map k a`
  erzeugt eine Map mit einem Element
- `insert :: Ord k => k -> a -> Map k a -> Map k a`
  fügt einen neuen Schlüssel und einen neuen Wert in die Map ein. Sollte der Schlüssel bereits bestehen, so wird der zugehörige Wert überschrieben
- `mapWithKey :: (k -> a -> b) -> Map k a -> Map k b`
  mappt eine Funktion über alle Werte der Map
- `filterWithKey :: Ord k => (k -> a -> Bool) -> Map k a -> Map k a`
  entfernt alle Werte aus der Liste, die das Prädikat (die gegebene Funktion) nicht erfüllen
- `insertWith :: Ord k => (a -> a -> a) -> k -> a -> Map k a -> Map k a`
  das Paar (Schlüssel, Wert wird in die Map eingefügt, falls der Schlüssel noch nicht existiert; wenn der Schlüssel existiert, wird das Ergebnis der gegebenen Funktion – angewandt auf den neuen und den alten Wert – als Wert des Schlüssels eingesetzt.
- `unionWith :: Ord k => (a -> a -> a) -> Map k a -> Map k a -> Map k a`
  vereinigt zwei Maps mittels einer Kombinierungsfunktion

3.5.1 mapPerKey

Der Eingabetyp der ersten Phase, `mapPerKey`, ergibt sich aus dem Typ der Eingabe von `mapReduce`. Diese Phase wendet die Funktion *map* auf jedes Element der Eingabe an. Aus dem Typ von *map* ergibt sich der Ergebnistyp für jedes Element der Eingabe: eine Liste von Zwischen-Schlüssel/Wert-Paaren.

```
mapPerKey :: Map k1 v1   -- Ein Schlüssel-zu-Eingabewert-Map
          -> [(k2, v2)]  -- Die Zwischen-Schlüssel/Wert-Paare
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
```

In dieser Phase wird zunächst die Schlüssel/Wert-Map in eine Liste von Schlüssel/Wert-Paaren verwandelt und hierauf das normale `map` mit der Funktion *mapper* angewandt. Um die Funktion *mapper*, die zwei Argumente nacheinander erwartet, auf ein Paar von Argumente anwenden zu können, wird sie unge*curry*t. Danach wird die entstehende Liste von Listen zu einer Liste konkateniert.

3.5.2 groupByKey

Der Ergebnistyp von `mapPerKey` liefert den Eingabetyp für `groupByKey`. Diese Phase gruppiert alle Zwischenwerte mit demselben Zwischenschlüssel, was eine Map als Ergebnistyp nahelegt.

```
groupByKey :: [(k2, v2)]   -- Die Zwischen-Schlüssel/Wert-Paare
           -> Map k2 [v2]  -- Die gruppierten Zwischenwerte
groupByKey = foldl insert empty
  where insert dict (k2, v2) = insertWith (++) k2 [v2] dict
```

In dieser Phase wir das Gruppieren durch die Konstruktion einer Map erreicht, die Schlüssel an zugehörige Werte bindet. Jedes einzelne Zwischen-Schlüssel/Wert-Paar wird folgendermaßen in die Map eingefügt: Wenn der gegebene Schlüssel noch nicht an irgendwelche Werte gebunden ist, dann wird ein neuer Eintrag mit einer einelementigen Liste erstellt. Andernfalls wird die einelementige Liste an die bereits eingefügten Werte angehängt. Diese Iteration über die Schlüssel/Wert-Paare wird durch einen `fold` ausgedrückt.

3.5.3 reducePerKey

Der Typ von `reducePerKey` ist ausreichend durch seine Position in der Funktionskomposition beschränkt. Sein Eingabetyp stimmt mit dem Ausgabetyp von `groupByKey` überein, sein Ausgabetyp stimmt mit dem Ausgabetyp von `mapReduce` überein:

```
reducePerKey :: Map k2 [v2]   -- Die gruppierten Zwischenwerte
             -> Map k2 v3     -- Eine Schlüssel-zu-Ausgabewert-Mapping
reducePerKey
  = mapWithKey unJust         -- 3. Verwandle Typ um Maybe zu entfernen
    . filterWithKey isJust    -- 2. Entferne Einträge mit Wert Nothing
    . mapWithKey reducer      -- 1. Wende reducer auf jeden 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-opt. Typ
```

In dieser Phase wird die Funktion *reducer* auf die Gruppen der Zwischenwerte gemappt während der Schlüssel jeder Gruppe erhalten wird. Vorher werden Einträge mit dem Wert `Nothing` entfernt.

3.6 Wörter-Zähl-Beispiel

Um die Häufigkeit von Wörtern in Dokumenten zu berechnen, eignet sich folgender Code:

```
wordOccurrenceCount = mapReduce mapper reducer
  where
    mapper  = const (map (flip (,) 1) . words) -- jedes Wort zählt als 1
    reducer = const (Just . sum)               -- berechne die Summe
                                               -- aller Zählungen
```

Das entstehende Programm kann man dann auf eine geeignet erstellte `Map` von Dokumenten-ID/Text-Paaren anwenden:

```
wordOccurrenceCount (fromList
  [("t01","ihr seid alle individuen"),
   ("t02","wir sind alle individuen"),
   ("t03","ich nicht")])
 ==
 fromList [("alle",2),("ich",1),("ihr",1),("individuen",2),("nicht",1),
           ("seid",1),("sind",1),("wir",1)]
```

Diese Spezifikation wird nun um Parallelität erweitert.

3.7 Parallelität im `MapReduce-Modell`

Für den Programmierer ist der Umgang mit Parallelität und der Verteilung der Daten auf die einzelnen Knoten meist transparent. Das Programmiermodel erlaubt von sich aus Parallelität. Die Implementierung regelt die komplexen Details wie zum Beispiel Lastverteilung, Netzwerkperformanz und Fehlertoleranz. Der Programmierer gibt nur noch bestimmte Parameter an, um z.B. die Anzahl der Reduce-Tasks festzulegen.

Im Weiteren werden zunächst die Möglichkeiten für eine Parallelisierung in einer verteilten Ausführung einer `MapReduce`-Berechnung herausgestellt. Danach wird das bis jetzt erarbeitete Modell um diese Parallelität erweitert.

3.7.1 Paralleler `map` über die Eingabedaten

Die Eingabedaten werden einzeln als Schlüssel/Wert-Paare verarbeitet. Die Anwendung von `map` auf eine Liste kann in totaler Datenparallelität ausgeführt werden, indem auf feinster Granularitätsstufe das `map` im Parallelen auf jedes einzelne Element angewandt wird. Dazu muss die Funktion, die an `map` übergeben wird, eine pure Funktion sein. Somit hat die Reihenfolge, in der die Schlüssel/Wert-Paare verarbeitet werden, keine Auswirkung auf das Ergebnis der `map`-Phase. Auf Kommunikation zwischen den einzelnen Threads kann dann verzichtet werden.

3.7.2 Paralleles Gruppieren der Zwischendaten

Das Gruppieren der Zwischendaten anhand des Schlüssels ist im Wesentlichen ein Sortierproblem. Dafür existieren verschiedene parallele Sortiermodelle. Bei einer verteilten `map`-Phase kann das Gruppieren auch mit dem verteilten `map` verbunden werden. Das Gruppieren kann also für jeden Bruchteil von Zwischendaten durchgeführt werden und die Ergebnisse des verteilten Gruppierens können zentral zusammengefügt werden, wie im Falle einer Parallel-Merge-All-Strategie (siehe [DGS+90]).

3.7.3 Paralleler `map` über Gruppen

Die Reduktion wird einzeln für jede Gruppe (ein Schlüssel mit einer Liste von Werten) durchgeführt. Hier greift wiederum das Muster des `map` auf eine Liste, deshalb ist hier totale Datenparallelität erlaubt wie im Falle der `map`-Phase.

3.7.4 Parallele Reduktion pro Gruppe

Unter der Annahme, dass die Reduktionsfunktion eine *eigentliche Reduktion* ist (siehe Abschnitt 3.2 auf Seite 8), kann jede Anwendung der Reduktion massiv parallelisiert werden. Dazu wird die Berechnung von Unter-Reduktionen in einer Baum-Struktur durchgeführt, indem die assoziative Operation auf die Knoten angewendet wird. Wenn die zweistellige Operation auch kommutativ ist, kann die Reihenfolge der Zusammenfassung der Ergebnisse willkürlich festgelegt werden.

3.7.5 Verfeinerung mit `combiner`-Funktion

Aus der Betrachtung der Verteilungsstrategie der Daten in Googles `MapReduce`-Modell wird klar, dass die Haupt-Herausforderung bezüglich der Performanz der verteilten Bearbeitung in der beschränkten Bandbreite des Netzwerks liegt. Deshalb wird besonderer Wert auf Datenlokalität gelegt. Zum einen werden die durch die Map-Tasks zu bearbeitenden Teildaten der Eingabedaten durch das verteilte Dateisystem möglichst so auf die lokalen Speicher der Map-Tasks verteilt, dass jeder Map-Task nur diese lokalen Daten für seine Berechnung benötigt. Um zum anderen die Menge der Zwischendaten zu reduzieren, die von den Map-Tasks zu den Reduce-Tasks geschickt werden, sollte eine lokale Reduktion durchgeführt werden, ehe kommuniziert wird. Beispielsweise entstehen bei dem eingangs erwähnten Problems, die Häufigkeiten von Wörtern in Dokumenten zu bestimmten, für häufig vorkommende Wörter (wie zum Beispiel Artikel) viele Paare mit dem Wort als Schlüssel und der Zahl 1 als Wert. Würden alle diese Zwischendaten verschickt, würde dies unnötig Bandbreite verbrauchen. Diese Zwischendaten könnten bereits von dem Map-Task geeignet reduziert werden.

Hierzu wird eine neue Eingabefunktion namens *combiner* eingeführt, „die eine teilweise Zusammenfassung dieser Daten durchführt, bevor diese über das Netzwerk verschickt werden. [...] Typischerweise wird der gleiche Code verwendet, um die Combiner- und Reduktionsfunktionen zu implementieren.“ [DG04, Übersetzung: Peter Lutz] Wenn beide Funktionen diesselbe *eigentliche Reduktion* (siehe Abschnitt 3.2 auf Seite 8) implementieren, dann fördert diese Verfeinerung idealerweise die Möglichkeit für massive Parallelität der Reduktion von Gruppen per Schlüssel. Hierbei hat die Baumstruktur für parallele Reduktion die Tiefe 2, wobei die Blätter der lokalen Reduktion entsprechen. Diese Verfeinerung dient einzig und allein dazu, die Menge der zu übertragenden Daten zu reduzieren. Auf die Anzahl der Prozessoren hat dies keine Auswirkungen.

3.7.6 Spezifikation

Mit dem *combiner* ergibt sich folgende verfeinerte parallele Spezifikation, die allerdings nicht auf das Task-Scheduling und die Benutzung eines verteilten Dateisystems (mit redundantem und verteiltem Speicher) eingeht. Auch wird die Aufteilung der Eingabedaten und das Zusammenfügen der Ausgabedaten als gegeben angenommen. Diese Spezifikation erfordert durch den Benutzer die explizite Bestimmung der Anzahl der Reduce-Tasks und der Partitionierungsfunktion, die auf der Schlüsselmenge der Zwischendaten arbeitet.

```
mapReduce  ::  Ord k2 =>
               Int                          -- Anzahl der Partitionen
           ->  (k2 -> Int)                  -- Partitionierung mittels Schlüssel
           ->  (k1 -> v1 -> [(k2,v2)])      -- die mapper-Funktion
           ->  (k2 -> [v2] -> Maybe v3)     -- die combiner-Funktion
           ->  (k2 -> [v3] -> Maybe v4)     -- die reducer-Funktion
           ->  [Map k1 v1]                  -- Verteilte Eingabe-Daten
           ->  [Map k2 v4]                  -- Verteilte Ausgabe-Daten

mapReduce parts keycode mapper combiner reducer
 =
     map (
         reducePerKey reducer         -- 7. Wende reducer auf jede Partitionierung an
       . mergeByKey )                 -- 6. Füge verteilte Zwischendaten zusammen
   . transpose                        -- 5. Transponiere verteilte Partitionierungen
   . map (
       map (
           reducePerKey combiner      -- 4. Wende combiner lokal an
         . groupByKey )               -- 3. Gruppiere lokale Zwischendaten
     . partition parts keycode        -- 2. Partitioniere lokale Zwischendaten
     . mapPerKey mapper )             -- 1. Wende mapper lokal auf jedes Stück an
```

Diese neue Funktion nimmt eine Liste von Maps als Eingabe, die den Teildaten der verschiedenen Map-Tasks entsprechen, und liefert eine Liste von Maps als Ausgabe zurück, die den Reduktionsergebnissen der verschiedenen Reduktions-Tasks entsprechen.

Das Argument `parts` definiert die Anzahl der Partitionierungen der Zwischendaten (diese entspricht der Anzahl der Reduktions-Tasks).

Das Argument `keycode` definiert die Partitionierungsfunktion auf der Schlüsselmenge der Zwischendaten. Diese Funktion bildet Schlüssel auf die Menge $\{1, \ldots, \texttt{parts}\}$ ab.

Das Argument `combiner` definiert die Combiner-Funktion für die Reduktion der Zwischendaten pro Map-Task. Sie hat denselben Typ wie `reducer`, allerdings wird aus Gründen der Generalisierung eine zusätzliche Typunterscheidung eingeführt. Der Ergebnistyp von `combiner` ist der Elementtyp reduziert durch `reducer`. Das `combiner`-Argument ist im Grunde optional, da es trivialerweise so definiert werden kann, als ob es alle Werte an `reducer` weiterreicht.

Die äußersten Anwendungen des `map` entsprechen den parallelen Map- und Reduce-Tasks einschließlich der Gruppierungs- und Zusammenfügungstätigkeiten auf den lokalen Daten. Dazwischen findet eine Transposition der Daten statt. Sie modelliert abstrakt die Kommunikation zwischen Map- und Reduce-Tasks. Für jede gegebene lokale Partition

werden also die verteilten physischen Anteile der Map-Tasks vereinigt, um die Eingabe für den Reduce-Task zu bilden, der für die Partitionierung verantwortlich ist.

Wegen des zusätzlichen Gebrauchs der Combiner-Funktion gibt es jetzt zwei Anwendungen von `reducePerKey`: eine für den Map-Task und eine für den Reduce-Task. Die zwei verwendeten Hilfsfunktionen werden folgendermaßen definiert:

```
-- Partitioniere Zwischendaten
partition :: Int -> (k2 -> Int) -> [(k2,v2)] -> [[(k2,v2)]]
partition parts keycode pairs = map select keys
 where
  keys        = [1..parts]          -- Die Liste mit {1, ..., parts}
  select part = filter pred pairs   -- Filtere Paare mittels Schlüssels
   where
    pred (k,_) = keycode k == part
```

Die Funktion `partition` erstellt eine verschachtelte Liste, wobei die inneren Listen den Partitionen der Eingabe entsprechen.

```
-- Füge Zwischendaten zusammen
mergeByKey :: Ord k2 => [Map k2 v3] -> Map k2 [v3]
mergeByKey =
    unionsWith (++)                       -- 2. Füge Maps zusammen
  . map (mapWithKey (const singleton))    -- 1. Migriere auf Listentyp
 where
  singleton x = [x]
```

Die Funktion `mergeByKey` fügt im Wesentlichen Maps zusammen, indem eine Liste von Werten für jeden Schlüssel gebildet wird. Das Zusammenfügen ist deutlich effizienter als eine generelle Gruppierungsoperation (wie zum Beispiel eine Sortierungsoperation), da alle eingehenden Maps für die Merge-Funktion bereits gruppiert und sortiert sind.

Eine weitere parallele Implementierung von `MapReduce` findet sich im nächsten Abschnitt.

4 Was ist Eden?

> „Eden erweitert die nicht-strikte funktionale Sprache Haskell um Konstrukte zur Kontrolle der parallelen Auswertung von Prozessen.“ [LOmPm05, Übersetzung: Peter Lutz]

Eden kümmert sich – transparent für den Benutzer – um die Kommunikation und die Synchronisation. Es ist als parallele funktionale Sprache geeignet, anspruchsvolle Skelette zu entwickeln. [LOmPm05, Phi]

4.1 Paralleles `MapReduce` in Eden

Komposition von `map` und `reduce` Die Kombinatoren `map` und `reduce` können folgendermaßen kombiniert werden:

```
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)
```

Je nach Richtung der Reduktion ändert sich dabei der Typ von *reduce*. Für parallele Implementierungen müssen die Typen `b` und `c` übereinstimmen und in den getrennten Unter-Reduktionen Assoziativität gelten. Außerdem sollte der Parameter `n` das neutrale Element von *reduce* sein. Wenn diese Bedingungen gelten, dann ist die Reduktionsrichtung irrelevant, da beide Berechnungen das gleiche ergeben. Falls *reduce* außerdem kommutativ ist, kann die parallele Implementierung die Eingabe umordnen.

Unter der Annahme, dass Assoziativität und Kommutativität für *reduce* gegeben sind, kann ein paralleles `MapReduce`-Skelett in Eden folgendermaßen definiert werden:

```
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 den Eingabestrom ...
unshuffle n list = ...                -- ... per round-robin auf np Ströme
spawn :: [Process a b] -> [a] -> [b]  -- erzeugt eine Menge von Prozessen ...
spawn ps inputs = ...                 -- ... mit zugehörigen Eingaben
```

Der Eingabestrom wird per *round-robin*-Methode auf `np` Eingaben für `np` Eden-Prozesse verteilt, die durch die Eden-Funktion `spawn` instanziiert werden. `Process` ist der Typkonstruktor für die Prozessabstraktionen von Eden, die mittels der Funktion

```
process :: (a -> b) -> Process a b
```

erzeugt werden. Die Typklasse `Trans` bietet implizit benutzte Funktionen für den Datenaustausch. Die instanziierten Prozesse führen eine Transformation mittels `map` aus und vorreduzieren die Ergebnisse durch die strikte linksassoziative Reduktion `foldl'`, wobei das gegebene neutrale Element auf die Prozesse dupliziert wird.

5 Fazit

Funktionale Skelette stellen neben Design Patterns und Refactorings eine weitere wichtige Säule des anspruchsvollen Software Engineerings dar, werden aber immer noch zu wenig beachtet. Googles `MapReduce`-Ansatz hat hier viel Aufmerksamkeit für die Programmierung mit Skelette geschaffen. In der Popularität von `MapReduce` liegt aber auch eine gewisse Gefahr. Wenn man ein konkretes Problem nicht gründlich genug analysiert hat und ein Skelett gewählt wurde, dass zu generell für das Problem ist, sind meist Performanzeinbrüche zu beklagen. Funktionale Skelette scheinen in ihrer Anwendung sehr bequem, das Problem muss aber bereits ausreichend analysiert worden sein, um das richtige Skelett wählen zu können.

Auch kann nicht jedes Problem in ein Funktionales Skelett gepresst werden, obwohl durch eine Hintereinanderausführung oder Komposition eines oder mehrerer Skelette eine große Anzahl an Problemen durch Skelette gelöst werden können. Um eine Performanzsteigerung von Berechnungen, wie Cole sie sieht, zu erreichen, müssten für die Skelette auf möglichst vielen Systemen optimierte Implementierungen angeboten werden. Vor allem im Bereich der Echtzeitanwendungen auf eingebetteten Systemen dürfte dies allerdings schwierig werden.

Das Thema ist dennoch ein interessantes und spannendes Forschungsfeld und wird dies voraussichtlich auch noch eine ganze Weile bleiben.

Literatur

[BDL09] Berthold, Jost ; Dieterle, Mischa ; Loogen, Rita: Implementing Parallel Google Map-Reduce in Eden. In: Sips, Henk (Hrsg.) ; Epema, Dick (Hrsg.) ; Lin, Hai-Xiang (Hrsg.): *Euro-Par 2009 Parallel Processing* Bd. 5704, Springer Berlin / Heidelberg, 2009 (Lecture Notes in Computer Science), 990-1002

[Col91] Cole, Murray: *Algorithmic skeletons: structured management of parallel computation*. Cambridge, MA, USA : MIT Press, 1991. – ISBN 0–262–53086–4

[Col04] Cole, Murray: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. In: *Parallel Comput.* 30 (2004), March, 389–406. http://dx.doi.org/10.1016/j.parco.2003.12.002. – ISSN 0167–8191

[DG04] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: simplified data processing on large clusters. In: *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*. Berkeley, CA, USA : USENIX Association, 2004, 10–10

[DG08] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: simplified data processing on large clusters. In: *Commun. ACM* 51 (2008), January, 107–113. http://doi.acm.org/10.1145/1327452.1327492

[DGS+90] Dewitt, D. J. ; Ghandeharizadeh, S. ; Schneider, D. A. ; Bricker, A. ; Hsiao, H. I. ; Rasmussen, R.: The Gamma Database Machine Project. In: *IEEE Trans. on Knowl. and Data Eng.* 2 (1990), March, 44–62. http://dx.doi.org/10.1109/69.50905. – ISSN 1041–4347

[has] 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, Abruf: 12. Dezember 2010

[Läm08] Lämmel, Ralf: Google's MapReduce programming model – Revisited. In: *Science of Computer Programming* 70 (2008), Nr. 1, 1 - 30. http://dx.doi.org/10.1016/j.scico.2007.07.001

[LOmPm05] Loogen, Rita ; Ortega-mallén, Yolanda ; Peña-marí, Ricardo: Parallel functional programming in Eden. In: *J. Funct. Program.* 15 (2005), May, 431–475. http://dx.doi.org/10.1017/S0956796805005526. – ISSN 0956–7968

[Phi] Philipps-Universität Marburg: *Eden: Parallel Functional Programming with Haskell*. http://www.mathematik.uni-marburg.de/~eden/, Abruf: 12. Dezember 2010

[Ste90] Steele, Guy L. Jr.: *Common LISP: the language (2nd ed.)*. Newton, MA, USA : Digital Press, 1990 http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html. – ISBN 1–55558–041–6