UNIVERSITÄT
KOBLENZ · LANDAU
Institute for Computer Science

Algebraic Analysis of MapReduce Samples

Bachelor Thesis

in partial fulfillment of the requirements for the degree of
Bachelor of Science (B.Sc.)
in Computer Sciences

submitted by
Andreas Brandt

First Advisor: Prof. Dr. Ralf Läemmel
Institute for Computer Science

Second Advisor: Vadim Zaytsev
Institute for Computer Science

Koblenz, February 2010

Statutory Declaration

I declare that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from these sources are clearly marked as such. This thesis was not submitted in the same or in a substantially similar version, not even partially, to any other authority to achieve an academic grading and was not published elsewhere.

. . . .

(Location, Date) (Signature)

Abstract

The Google MapReduce framework has been implemented in several programming languages like C++, Java, Python and other. They cover a range of use cases from simple wordcount examples to complex web indexing. We inspect the internals of a number of these use cases and identify common patterns among them. Especially interesting finds are the use of monoidal structures as these provide great opportunities for parallel programming. Beyond these examples we give a brief overview of monoids that possibly can be used in a MapReduce environment. Finally we compare some characteristics of the examined frameworks.

Our programming language of choice for implementing example use cases with a monoidal abstraction layer is Haskell. As a pure funcional language it has no side effects and thus is a perfect tool for parallel programming. Combined with its powerful type system, which also includes a specific type class "Monoid", a concise implementation of common use cases for the MapReduce framework is possible.

Contents

1 Introduction **3**
1.1 Monoids and MapReduce . . . 3

2 Survey of MapReduce frameworks **4**
2.1 Google's MapReduce . . . 4
2.1.1 WordCount . . . 4
2.1.2 Distributed Grep . . . 4
2.1.3 Distributed Sort . . . 5
2.1.4 Count of URL Access Frequency . . . 6
2.1.5 Reverse Web-Link Graph . . . 6
2.1.6 Term-Vector per Host . . . 7
2.1.7 Inverted Index . . . 8
2.2 Hadoop . . . 9
2.2.1 WordCount . . . 9
2.2.2 Grep . . . 10
2.2.3 PiEstimator . . . 10
2.2.4 Sort . . . 11
2.2.5 Sudoku . . . 12
2.2.6 Pentomino . . . 13
2.3 Real World Haskell . . . 15
2.3.1 Counting Lines . . . 15
2.3.2 Finding the Most Popular URLs . . . 15
2.4 Holumbus Framework . . . 17
2.4.1 WordCount . . . 17
2.4.2 Grep . . . 17
2.4.3 Sort . . . 17
2.4.4 Crawler . . . 18
2.5 Disco . . . 23
2.5.1 Wordcount . . . 23
2.6 Qt Concurrent . . . 25
2.6.1 Wordcount . . . 25
2.7 Skynet . . . 26
2.7.1 Grep . . . 26
2.8 Sawzall . . . 27
2.8.1 Collection . . . 27
2.8.2 Sample . . . 27
2.8.3 Sum . . . 27
2.8.4 Maximum . . . 27
2.8.5 Quantile, Top and Unique . . . 28
2.9 Hadoop and aggregators . . . 29
2.9.1 AggregateWordCount . . . 29
2.10 Comparison Of MapReduce Frameworks . . . 31

2.11 Further examples of monoids 33

3 Implementation **36**

3.1 Built-in monoids 36

3.2 Non-standard monoids 38

3.2.1 Monoids over `Maps` 39

3.3 Non-Trivial Monoids/Reducers 40

4 Conclusion **41**

1 Introduction

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. [9]

Sawzall [20] is an enhancement to MapReduce as it allows to use predefined *aggregators* without the need to specify both a map and reduce function. Ralf Lämmel [17] reverse-engineered the MapReduce and Sawzall papers to design an executable specification. He contended that the essence of a Sawzall program is to identify the characteristic arguments of a list homomorphism: a function to be mapped over the list elements as well as the monoid to be used for the reduction. This paper aims to show whether this monoidal approach holds true for common use cases of MapReduce based on a variety of examples.

1.1 Monoids and MapReduce

A monoid is a set with an associative binary operation with an identity element[22]. Or, in terms of halfgroups: a monoid H is a triple $H = (M, \circ, 1_H)$ of a halfgroup $(M, \circ)$ and an identity element 1_H[10]. For example the set of natural numbers in combination with addition as binary operation and 1 as identity element form a monoid which we denote as $(\mathbb{Z}, +, 1)$. If we add commutativity, the monoid is called a commutative monoid which we refer to simply as monoid throughout this paper.

We want to show that the reduce function for all popular examples of the MapReduce framework can be written in terms of monoids.

First, if the querying operations are commutative across records, the order in which the records are processed is unimportant. We can therefore work through the input in arbitrary order. Second, if the aggregation operations are commutative, the order in which the intermediate values are processed is unimportant. Moreover, if they are also associative, the intermediate values can be grouped arbitrarily or even aggregated in stages. As an example, counting involves addition, which is guaranteed to be the same independent of the order in which the values are added and independent of any intermediate subtotals that are formed to coalesce intermediate values. [20]

2 Survey of MapReduce frameworks

In this survey we will look at several open source MapReduce frameworks and examine a couple of sample use cases. In particular we try to reduce these use cases to a monoidal structure and identify the underlying monoid.

2.1 Google's MapReduce

2.1.1 WordCount

WordCount is the "Hello World" of MapReduce-Frameworks and is also described in the original MapReduce paper [9]. This program counts the occurrence of a specific word in a set of distributed documents. $\mathcal{MAP}$ emits each word plus an associated count of occurrences which would be `1` for each match whereas $\mathcal{REDUCE}$ adds up all emitted numbers for a particular word. In terms of $\langle key, value \rangle$ pairs, $\mathcal{MAP}$ generates following output:

```
<"word1", 1>
<"word2", 1>
<"word2", 1>
<"word3", 1>
...
```

$\mathcal{REDUCE}$ would then take all $\langle key, value \rangle$ pairs of a specific key and aggregate the corresponding values. A reducer for `word2` for example would emit $\langle "word2", 2 \rangle$.

We can represent $\langle key, value \rangle$ pairs with the dictionary type Data.Map.Map in Haskell. Each $\mathcal{MAP}$ emits several Maps, each containing a single $\langle word, count \rangle$ pair with `count` always being `1`. In the reduce phase all values for each key need to be summed up. The Haskell function unionWith (+) does exactly this for two Maps. This is the binary, associative function we need to form a monoid. The other precondition for a monoid needs to be checked: the existence of an identity element. In this case the identity element is just an empty Map.

2.1.2 Distributed Grep

$\mathcal{MAP}$ emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output [9]. The distributed grep program searches for a character pattern and writes intermediate data only when a match is encountered [12].

This is a very brief description and it is unlikely that the map function really only emits a line instead of a $\langle key, value \rangle$ pair like in all the other examples from the MapReduce paper. If it was that simple the monoid would be the concatenation of the lines with the empty string as identity element. We will focus on a solution which involves $\langle key, value \rangle$ pairs.

If we want to do a distributed grep for "hello" as an example, the map phase will generate $\langle key, value \rangle$ pairs like this:

```
<"", "hello world\n">
<"", "Say hello to Jim!\n">
...
```

Again, we use a Map as datatype. The key is not needed here because there is only one key: the string "hello". As result of $\mathcal{REDUCE}$ we expect a Map of all lines which contain the word "hello". This is just a matter of concatenating the unchanged intermediate values. One observation we make is concatenation not being commutative. This means we cannot be sure to get the same result with the same input every time we run a distributed grep. For a grep this is not important as it only matters what the matches are and not in which order they are presented. We could circumvent this inconvenience by adding a sort function of some kind. Nonetheless we will see the distributed grep still forms monoid.

We denote concatenation as $(++)$ like the Haskell operator for list concatenation and use unionWith (++) to concatenate the values of two Maps. This forms the associative operation of the monoid. The neutral element is an empty Map.

2.1.3 Distributed Sort

The map function extracts the key from each record, and emits a $\langle key, record \rangle$ pair. The reduce function emits all pairs unchanged [9]. This computation depends on the partitioning facilities and ordering properties described in Google's MapReduce paper.

Google's MapReduce performs automatic sorting:
We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted [9].

This is achieved by using a sorting key:
A three-line Map function extracts a 10-byte sorting key from a text line and emits the key and the original text line as the intermediate key/value pair. We used a built-in Identity function as the Reduce operator. This function passes the intermediate key/value pair unchanged as the output key/value pair [9].

If we did not want to utilize a specific framework but specify $\mathcal{MAP}$ and $\mathcal{REDUCE}$ for a generic MapReduce model we would split the input in several chunks of data and sort every chunk within the map function and then pass the ordered sets to the reduce function (see figure 1) which would merge these sets as described in [8]. They suggest following scheme to do a distributed sort with MapReduce:

Map Use a range partitioner in mappers, so that records are partitioned into ordered buckets, each is over a mutually exclusive key range and is designated to one reducer.

Reduce For each Map-Reduce lineage, a reducer reads the designated buckets from all the mappers. Data in these buckets are then merged into a sorted set. This sorting procedure can be done completely at the reducer side, if necessary, through an external sort. Or, mappers can sort data in each buckets before sending them to reducers. Reducers can then just do the merge part of the merge sort using a priority queue.

Aswell as a third `Merge` phase to merge the results of several reducers. $\mathcal{MAP}$ will do the sorting procedure and emits $\langle key, value\rangle$ pairs where `value` is an Ascending Set and `key` does not matter in a simple case with only one search. If we would do several distinct searches, perhaps on different parts of the initial data base, `key` would indicate for example the number of each search process.

Ralf Lämmel [17] already implemented a sorting monoid in Haskell. He uses ascending sets as input for $\mathcal{REDUCE}$ and defines a merge function as the binary operation for combining two ascending sets. The identity element is an empty ascending set.

Hence, a monoid for sorting is $(AscendingSet, merge, AscendingSet[])$.

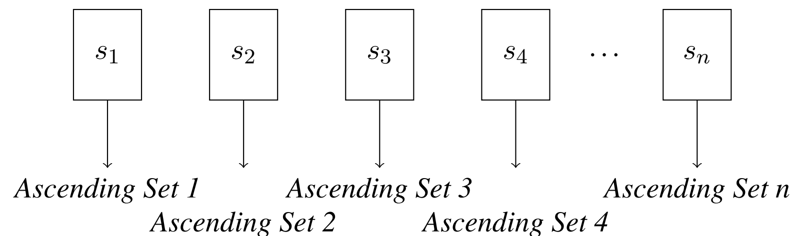


Figure 1: $\mathcal{MAP}$ generates ordered sets.

2.1.4 Count of URL Access Frequency

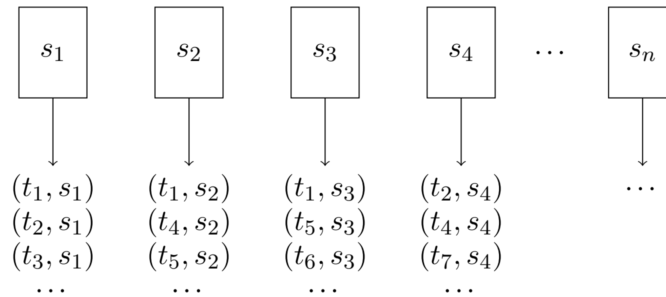
Count of URL Access Frequency: The map function processes logs of web page requests and outputs $\langle URL, 1\rangle$. The reduce function adds together all values for the same URL and emits a $\langle URL, totalcount\rangle$ pair [9].

This is basically a wordcount where the word is replaced with the URL so the same Monoid can be used as in the WordCount example in section 2.1.1.

2.1.5 Reverse Web-Link Graph

A forward web-link graph is a graph that has an edge from node URL1 to node URL2 if the web page found at URL1 has a hyperlink to URL2. A reverse web-

link graph is the same graph with the edges reversed. MapReduce can easily be used to construct a reverse web-link graph. [18] The map function outputs $\langle target, source\rangle$ pairs for each link to a target URL found in a page named `source`. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $\langle target, list(source)\rangle$. [9]



$\mathcal{MAP}$ essentially generates a list of $\langle target, source\rangle$ tuples for each source and aggregates them to $\langle target, list(source)\rangle$ pairs which serve as intermediate data and contain a list of Maps of all target URLs with their corresponding source URLs.

$\mathcal{REDUCE}$ only accepts the data for a specific key before it combines the output. We can concatenate the results like we did before in the grep example in section 2.1.2 (and use the same monoid again).

2.1.6 Term-Vector per Host

A term vector summarizes the most important words that occur in a document or a set of documents as a list of $\langle word, frequency\rangle$ pairs. The map function emits a $\langle hostname, termvector\rangle$ pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final $\langle hostname, termvector\rangle$ pair [9].

A term vector may look like this:

```
[("word1", 8), ("word2", 4), ("word3", 7)]
```

We use a Map to represent a term vector. $\mathcal{REDUCE}$ then receives a set of term vectors for a given host from several $\mathcal{MAP}$ instances:

```
("hostX", [("word1", 8), ("word2", 4), ("word3", 7)])
("hostX", [("word1", 3), ("word5", 6), ("word3", 3)])
("hostX", [("word1", 1), ("word6", 3), ("word2", 9)])
...
```

These $\langle host, TermVector \rangle$ pairs are also Maps. To aggregate Maps we used the function unionWith before. In the WordCount example we had simple integer values and used addition for aggregation. Now we have term vectors instead of integer values and need to find an associative operation for aggregation. Since term vectors are Maps and the value of them are integers, we can use unionWith (+).

So we have a merge function like this: unionWith (unionWith (+)). The inner unionWith adds up the term vectors and the outer aggregates the term vectors per host.

The identity element is again the empty Map. The implementation details are shown in section 3.

2.1.7 Inverted Index

$\mathcal{MAP}$ parses each document and emits a sequence of $\langle word, documentID \rangle$ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a $\langle word, list(documentID) \rangle$ pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions [9]. The Inverted Index is very similar to the Reverse Web-Link Graph we have seen in section 2.1.5. In fact we can replace the terms `target` and `source` with `word` and `document` we have exactly the same situation as in the example about the Reverse Web-Link Graph. Thus we can use the same monoid.

2.2 Hadoop

Hadoop is a framework for running applications on large clusters built of commodity hardware. The Hadoop framework transparently provides applications both reliability and data motion. Hadoop implements a computational paradigm named Map/Reduce, where the application is divided into many small fragments of work, each of which may be executed or reexecuted on any node in the cluster. In addition, it provides a distributed file system (HDFS) that stores data on the compute nodes, providing very high aggregate bandwidth across the cluster. Both Map/Reduce and the distributed file system are designed so that node failures are automatically handled by the framework.[1]

Amazon for example offers a MapReduce service[2] which is based on Hadoop. Customers can send their data to an "Amazon Bucket" and then execute an arbitrary MapReduce program written for the Hadoop framework on the Amazon infrastructure.

2.2.1 WordCount

Listing 1 shows the Hadoop example implementation of the WordCount reduce function in Java. It takes a key and a list of all associated values just like Google's MapReduce. The values which are associated with this key are passed to the reduce function as an Iterator. This association is done in the map function which already groups matching keys in a combine phase, again imitating the behaviour of Google's MapReduce. The framework calls this method for each $\langle key, (list\ of\ values)\rangle$ pair in the grouped inputs [2]. So the result contains all keys with the aggregated values per key. This reduce function does not return a $\langle key, value\rangle$ pair but stores the result in an OutputCollector as a side effect.

```
1 public void reduce(Text key, Iterator <IntWritable> values,
2                     OutputCollector<Text, IntWritable> output,
3                     Reporter reporter) throws IOException {
4   int sum = 0;
5   while (values.hasNext()) {
6     sum += values.next().get();
7   }
8   output.collect(key, new IntWritable(sum));
9 }
```

Listing 1: Reduce function for the WordCount example in Java

The underlying monoid is the same as in the Google WordCount example in section 2.1.1. $\mathcal{MAP}$ stores $\langle key, value\rangle$ pairs in the OutputCollector and $\mathcal{REDUCE}$ then aggregates the final mapping.

[1] http://wiki.apache.org/hadoop/

[2] http://aws.amazon.com/elasticmapreduce/

2.2.2 Grep

When we look at the Hadoop grep example it differs from the original definition in the Google MapReduce paper in some points. The map function (listing 2) does not emit pairs of $\langle pattern, matchedstring\rangle$ but $\langle matchedstring, 1\rangle$ pairs. This works because the pattern is not needed for performing the reduce function. Instead, this example adds extra functionality to grep by counting the occurrences of equal output lines in the reduce function.

```
1  public void map(K key, Text value,
2                  OutputCollector<Text, LongWritable> output,
3                  Reporter reporter) throws IOException {
4    String text = value.toString();
5    Matcher matcher = pattern.matcher(text);
6    while (matcher.find()) {
7      // group is 0 in this example
8      output.collect(new Text(matcher.group(group)), new LongWritable(1));
9    }
10 }
```

Listing 2: Map function for a distributed grep in Java

The reduce function (listing 3) looks almost *exactly* like the WordCount example and infact performs the same calculation. The identity function mentioned in Google's MapReduce paper has been replaced with a counting function. Furthermore the output is sorted by decreasing occurrences.

```
1  public void reduce(K key, Iterator<LongWritable> values,
2                     OutputCollector<K, LongWritable> output,
3                     Reporter reporter) throws IOException {
4    long sum = 0;
5    while (values.hasNext()) {
6      sum += values.next().get();
7    }
8    output.collect(key, new LongWritable(sum));
9  }
```

Listing 3: Reduce function for a distributed grep in Java

Thus, the monoid for reduction is the same as in the WordCount example in section 2.2.1.

2.2.3 PiEstimator

PiEstimator is an implementation of a monte carlo method [21] for estimating π. π is calculated from a large set of data which is generated with a statistical approach.

Given a square, a number of random coordinates within the square are generated. Then a corner arc is constructed as shown in figure 2 and the number of points within the corner arc divided by the number of points outside the corner arc yields an approximation of π. This approximation gets better the more coordinates are available due to the law of large numbers.

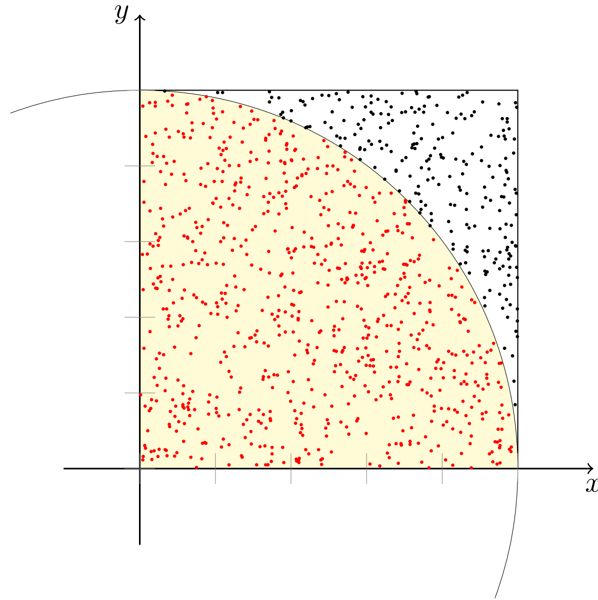


Figure 2: statistical approximation of π

In a MapReduce environment, $\mathcal{MAP}$ generates the random coordinates and checks whether they are within the boundaries of the corner arc. It then emits $\langle key, value \rangle$ pairs where `key` indicates wether a coordinate lies within the corner arc or not and `value` is always 1 to count the occurrences. The implementation of the Hadoop example generates several random numbers with each map instance.

$\mathcal{REDUCE}$ then needs to aggregate the `values` of each `key`. Listing 4 shows the Hadoop example which uses one reducer with a conditional branch to distinguish between the two `keys`.

In this case the OutputCollector is not used and the result is stored in the member variables numInside and numOutside; the Hadoop framework is not utilized as intended. Regardless, we can use the unionWith (+) function again to create a monoid for this operation. The result would be a Map with two `keys`, 0 and 1, with the corresponding values of numInside and numOutside.

2.2.4 Sort

Since sorting is part of the Hadoop framework and data is sorted automatically between $\mathcal{MAP}$ and $\mathcal{REDUCE}$ aswell after each $\mathcal{REDUCE}$ the implementation is

```java
public void reduce(LongWritable key,
                    Iterator <LongWritable> values,
                    OutputCollector<WritableComparable, Writable> output,
                    Reporter reporter) throws IOException {
  if (key.get() == 1) {
    while (values.hasNext()) {
      long num = values.next().get();
      numInside += num;
    }
  } else {
    while (values.hasNext()) {
      long num = values.next().get();
      numOutside += num;
    }
  }
}
```

Listing 4: Reduce function for π-estimation in Java

trivial and not different from the sorting example in Google's MapReduce: we just use an identity mapper and an identity reducer (listing 5). It writes all keys and values directly to the output.

```java
  public void reduce(K key, Iterator<V> values,
                     OutputCollector<K, V> output, Reporter reporter)
    throws IOException {
    while (values.hasNext()) {
      output.collect(key, values.next());
    }
  }
```

Listing 5: Identity reducer in Java

2.2.5 Sudoku

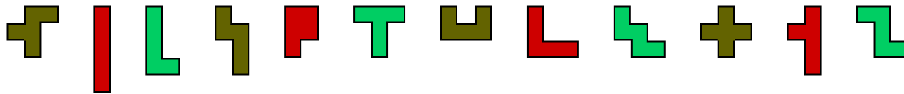
Sudoku solver using Knuth's Dancing Links algorithm [16].
The example from Hadoop is not a distributed Sudoku Solver:

> "The sudoku solver is so fast, I didn't bother making a distributed version."[2]

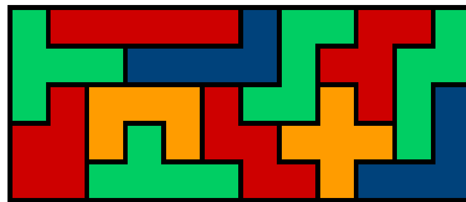
However, the next example, a pentomino solver, is also based on Knuth's dancing links algorithm and implemented in a distributed way so we can assume they both behave similar in a MapReduce environment.

2.2.6 Pentomino

The pentominoes are 12 pieces formed by joining five squares along their sides in all possible shapes. These are the 12 pentominoes:



The pentominoes can be used together to form different shapes.



Donald Knuth shows that this puzzle of forming a given shape with the 12 pentominoes is an example of a more general problem, known as the Exact Cover problem[16]. Therefore, we can take a general algorithm that solves the Exact Cover problem and use it to solve the pentomino puzzle.

Exact Cover Problem Given a matrix of 0's and 1's, the problem is to find an "exact cover" of the matrix's columns: a subset of its rows such that, put together, they contain exactly one 1 in each column of the matrix. The Exact Cover problem is known to be NP-complete[11]. Figure 2.2.6 shows a matrix which represents an Exact Cover problem. This matrix has only one solution: rows 1, 4, and 5.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Figure 3: A matrix representing an Exact Cover problem

An algorithm to solve the Exact Cover Problem is as follows:
1) Choose a column **c** from the matrix.
2) Choose, in turn, each row **r** where **c** contains a 1. For each **r** do the following:
3) Erase all columns where **r** contains a 1, and all rows that contain a 1 in any of those columns.
4) Apply the algorithm recursively to this reduced matrix.

If we choose the first column at step 1 we first try with row 2. Step 3 tells us to eliminate columns 1, 4, and 7, and rows 2, 4, 5, and 6. We are left with following matrix:

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

If we now choose the first column and eliminate all columns and rows according to the algorithm we are left with a matrix with only one column and no rows. This matrix has no solutions and we backtrack to the previous step and apply the algorithm until we have a solution.

Now we get to the relation between the pentomino puzzle and the Exact Cover problem. Construct a matrix with 72 columns: 12 for the twelve pentominoes and 60 for the cells of the board. Now, for each way a pentomino can be placed on the board, insert a row in the matrix. The row should contain a 1 in one of the first 12 columns, indicating the pentomino, and five 1's among the other 60 rows, indicating the five cells of the board the pentomino occupies; everywhere else the row should contain zeros. This should be done for all possible ways of placing each of the twelve pentominoes on the board. The matrix corresponding to a 8x8 square with a 2x2 hole in the middle, for example, has 1568 rows. Now the problem of finding a solution to the puzzle is equivalent to finding an exact cover of this matrix. [5]

Distributed pentomino solver We can see the opportunity for parallel computation: all branches of the same step are completely independent from each other. The question of granularity of parallel computation arises. The Hadoop example calculates *prefixes* which determine the work for the map function. The Hadoop documentation about the distributed pentomino solver [2]: It generates a complete list of prefixes of length N with each unique prefix as a separate line. A prefix is a sequence of N integers that denote the index of the row that is choosen for each column in order. Note that the next column is heuristically choosen by the solver, so it is dependant on the previous choice. That file is given as the input to map/reduce. The output key/value are the move prefix/solution as Text/Text.

Each map takes a line, which represents a prefix move and finds all of the solutions that start with that prefix. The output is the prefix as the key and the solution as the value. The map function breaks the prefix string into moves (a sequence of integer row ids that will be selected for each column in order) and finds all solutions with that prefix. The result of the map function is saved directly in a file and thus no reduce function is needed and the identity reducer is used.

2.3 Real World Haskell

MapReduce serves as an example for concurrent and multicore programming in Real World Haskell[19]. The authors refer to the implementation as a greatly simplified, but still useful Haskell equivalent to Google's MapReduce infrastructure. It comes without a file system and is executed in a multicore environment rather than in a distributed environment. It does not follow the classical MapReduce scheme as it uses different types for $\mathcal{MAP}$ and $\mathcal{REDUCE}$. $\mathcal{MAP}$ has the type `a -> b` as opposed to the classical `(k1, v1) -> [(k2, v2)]`, there is no key domain. In addition, strategies for evaluating $\mathcal{MAP}$ and $\mathcal{REDUCE}$ can be specified, for example to force evaluation to weak head normal form[3].

The task they chose as an example is the parallel processing of web server log files.

2.3.1 Counting Lines

This is just a trivial alteration of the WordCount example by choosing an end-of-line character as the word to be count. A notable difference is the specification of the evaluation strategy for the execution of $\mathcal{MAP}$ and $\mathcal{REDUCE}$ in parallel. `rnf` stands for reduce to normal form. Discussing parallel strategies is out of scope of this paper and will not be regarded further.

```
1 lineCount :: [LB.ByteString] -> Int64
2 lineCount = mapReduce
3               rnf (LB.count '\n')   -- Map
4               rnf sum               -- Reduce
```

Listing 6: Counting Lines with MapReduce

The lack of a key domain allows for a simpler monoid than in the previous WordCount examples. We do not need the `Map` datatype but can use simple **Integer**s. The monoid's associative operation is addition and the identity element is `0`.

2.3.2 Finding the Most Popular URLs

In this example, we count the number of times each URL is accessed. In the map phase we create a `Map` from a URL using the number of times it was accessed. In the reduce phase, we union-merge these maps into one [19]. The authors claim the example is from the Google's MapReduce paper, but as we will see it is a slightly modified version of the URL Access Frequency problem in section 2.1.4. $\mathcal{MAP}$ does not output $\langle URL, 1\rangle$ anymore but $\langle URL, count\rangle$, partially the counting of

[3] `http://en.wikibooks.org/wiki/Haskell/Graph_reduction#Weak_Head_Normal_Form`

the access frequency of a URL already takes place here (the values are *grouped by key*). The log file is split into several chunks and $\mathcal{MAP}$ counts the URLs of each chunk. For $\mathcal{REDUCE}$ there is following definition in Google's MapReduce paper:

> "The reduce function adds together all values for the same URL and emits a $\langle URL, totalcount\rangle$ pair."

The example from Real World Haskell adds together all values for *each* URL and emits a *list* of $\langle URL, totalcount\rangle$ pairs. Listing 7 shows both the map and reduce function of this example.

```
1  countURLs :: [L.ByteString] -> M.Map S.ByteString Int
2  countURLs = mapReduce
3                  rwhnf (foldl' augment M.empty . L.lines)  -- Map
4                  rwhnf (M.unionsWith (+))                  -- Reduce
5    where augment map line =
6              case match (compile pattern []) (strict line) [] of
7                Just (_:url:_) -> M.insertWith' (+) url 1 map
8                _ -> map
9          strict  = S.concat . L.toChunks
10         pattern = S.pack "\"(?: GET|POST|HEAD) ([^ ]+) HTTP/"
```

Listing 7: Finding the Most Popular URLs with MapReduce

The **import** statements are omitted for brevity. The L and S qualifiers indicate Haskell ByteStrings instead of **String** for efficiency, the M qualifier denotes the Haskell Data.Map module. countURLs takes a list of ByteStrings and performs $\mathcal{MAP}$ on each element of this list. The map function emits a Map with all found URLs as keys and the number of occurrences as value. $\mathcal{REDUCE}$ then merges all the Maps with unionsWith as seen in the source code.

The resulting monoid is $(Map, unionWith(+), empty)$. The binary operation is unionWith (+) and the identity element is the empty Map which is, in Haskell notation, just empty.

The monoid of Google's URLCount is the exactly the same, but it will only emit one $\langle URL, count\rangle$ pair per reducer instead of calculating all counts.

2.4 Holumbus Framework

Holumbus is a Haskell library which provides the basic building blocks for creating powerful indexing and search applications. This includes a framework for distributed crawling and indexing as well as distributed query processing. Additionally, a full-fledged distributed Map-Reduce framework is included. [4]

The Holumbus documentation covers a few examples including a wordcount, grep and sort implementation aswell as a web crawler.

2.4.1 WordCount

The Holumbus framework offers the option to omit the existence of a key domain in the reduce function. Each $\mathcal{MAP}$ creates a $\langle key, value\rangle$ pair (see listing 8 for an example). These pairs are partitioned and sent back to the master. The default partition function guarantees that all pairs with the same key are assigned to the same reducer machine. In the reduce phase each worker groups the key-value-pairs by their keys and then processes the reduce function [4]. The reduce function (list-

```
1 mapWordFrequency :: () -> String -> String -> IO [(String, Integer)]
2 mapWordFrequency _ _ v = return $ map (\s -> (s,1)) $ words v
```

Listing 8: Holumbus reduce function for WordCount

ing 9) discards the `key`, which is the second parameter, and returns the number of occurrences of the word wrapped in a **IO** (**Maybe Integer**) type. Since the ac-

```
1 reduceWordFrequency :: () -> String -> [Integer] -> IO (Maybe Integer)
2 reduceWordFrequency _ _ vs = return $ Just $ sum vs
```

Listing 9: Holumbus reduce function for WordCount

tual reducing only involves **sum** vs we will disregard the wrapping and assume the monoid over addition here like we did in section 2.3.1.

2.4.2 Grep

The reduce function used by the Holumbus grep is the `id` function and therefore uses the identity reducer as discussed before in section 2.2.2.

2.4.3 Sort

The implementation of a distributed sort is achieved by grouping the $\langle key, value\rangle$ pairs between two following phases which is provided by the Holumbus MapRe-

[4] `http://holumbus.fh-wedel.de/trac`

duce framework. Listing 10 shows a map function utilizing this mechanism to sort a list of names as example. The map function takes a list of tuples as value ele-

```
1 mapSort :: () -> () -> (String, String) -> IO [(String, (String, String))]
2 mapSort _ _ v = return $ [(snd v, v)]
```

Listing 10: Holumbus mapping function for sort in Haskell

ment. The first element of a tuple represents the forename while the second one the surname. The map function just creates a new $\langle key, value\rangle$ pair with the surname as key and the input tuple as value element.
The default partition function at the end of the map phase takes care of the correct splitting of the data among the reducer machines. The default merge function at the beginning of the reduce phase groups the name-pairs with the same surname. Another feature of this grouping which is crucial for this example, is that the keys are sorted in ascending order. The reduce function just returns the input values, similar to the id function of the Haskell standard library [4].

```
1 reduceSort :: () -> String -> [(String, String)] -> IO (Maybe [(String, String)])
2 reduceSort _ _ vs = return (Just vs)
```

Listing 11: Holumbus reduce function for sort in Haskell

Again, there is no use of the monoid of ascending sets as discussed in section 2.2.4 because of the built-in sorting mechanism of the framework which makes the reduce function trivial.

2.4.4 Crawler

The crawler has a global configuration state, called "crawler-state", which contains two lists. The first list stores the Uniform Resource Locators (URLs) of all websites which need to be crawled (todo-list) while the second one contains the URLs of all sites which were already processed (done-list). Unless the todo-list is not empty, the workers in the map phase load the requested websites, extract their links and save them on local disk. The processed URLs and the links are collected in the reduce phase and with the help of the previous todo- and done-lists the crawler-state is updated. This is only successful, if the reduce phase is processed on only one single worker. [4] [3]

The reduce function of the crawler receives both the todo-list and the done-list from the global crawler-state aswell as a list of links which were extracted from the currently processed website. These links are merged with the current todo-list and done-list so there are no duplicates and websites are not processed twice. The URL of the current website is stored in the done-list.

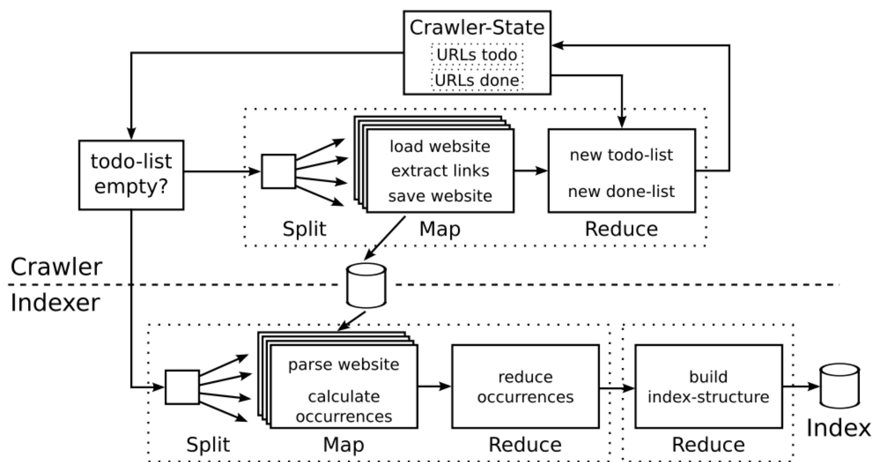


Figure 4: Scheme of the Holumbus crawler

The actual implementation is slightly more complicated. As we see from the signature from the corresponding reduce function in listing 13, it takes the whole crawler state as argument. To the data generated from the current website a MD5Hash is added to reference the document. The basic functionality of this function is a `fold`

```
1 processCrawlResults :: (HolDocuments d a, Binary a) =>
2                            CrawlerState d a
3                        -> ()
4                        -> [(MD5Hash, Maybe (Document a), S.Set URI)]
5                        -> IO (Maybe (CrawlerState d a))
6 processCrawlResults oldCs _ l = do
7   cs' <- foldM process oldCs l
8   return $ Just cs'
```

Listing 12: Reduce function for crawling in Haskell

to compute a new state from the old one. The **foldM** function is analogous to foldl, except that its result is encapsulated in a monad [5].

What is the binary operation and the identity element? **foldM** uses process (listing 13) for calculating the result so this should be the binary operation. The identity element must then satisfy the equation `process x id = process id x = x`. This equation cannot be satisfied because process takes two arguments of different types, thus the identity element would need to have both types which is impossible. Even if we dig further into the function it is not easy to spot a reducer monoid. The process function is not really aggregating values but updating

[5]http://www.haskell.org/onlinereport/monad.html

```haskell
process :: (HolDocuments d a, Binary a) =>
  CrawlerState d a ->
  (String, Maybe (Document a), S.Set URI) ->
  IO (CrawlerState d a)
process cs (theMD5, mdoc, refs)
  = if isJust mdoc then do
      if isJust (cs_docHashes cs) && M.member theMD5
                                              (fromJust $ cs_docHashes cs)
        then do
          let hashes = fromJust $ cs_docHashes cs
          let old    = fromJust $ M.lookup theMD5 hashes
          let new    = uri $ fromJust mdoc
          (newDocs, newHashes) <- update (cs_docs cs) hashes
                                                      theMD5 old new
          return cs { cs_docs = newDocs
                    , cs_toBeProcessed = S.union (cs_toBeProcessed cs)
                            (S.difference (S. filter (cs_fCrawlFilter cs) refs)
                                          (cs_wereProcessed cs))
                    , cs_docHashes = Just $ newHashes }
        else do return $
            cs { cs_toBeProcessed = S.union (cs_toBeProcessed cs)
                    (S.difference (S. filter (cs_fCrawlFilter cs) refs)
                                  (cs_wereProcessed cs))
               , cs_docs      = snd (insertDoc (cs_docs cs) (fromJust mdoc))
               , cs_docHashes = if isJust (cs_docHashes cs)
                                then Just $ M.insert theMD5
                                              (uri $ fromJust mdoc)
                                              (fromJust $ cs_docHashes cs)
                                else Nothing }
    else return cs
```

Listing 13: process function

the `CrawlerState`. For each document the state is immediately updated. Traditional aggregating would mean to collect all values (actual reducing task) and after that update the state with the gathered result. We could try to refactor the second argument of process so it is not a triplet anymore but also a CrawlerState. If we look at the structure of CrawlerState (see listing 14) we can spot some similarities between the crawler state and the triplet (**String**, **Maybe** (Document a), S.Set URI).

The **String** of the triplet is related to cs_docHashes, it is the MD5 hash of the current document. We could write it as **Just** (fromList [(theMD5, mdoc)]) and extract it again when needed. The second value, **Maybe** (Document a), is related to cs_docs and represents the current website. We do not need the **Maybe** wrapper if we check the existence of the website in the map function. Afer all the map function should never emit a value for a non-existing website. Finally the third value, S.Set URI, represents cs_toBeProcessed. These is a set of all links found

```
data CrawlerState d a
    = CrawlerState
      { cs_toBeProcessed   :: S.Set URI
      , cs_wereProcessed   :: S.Set URI
      , cs_docHashes       :: Maybe (M.Map MD5Hash URI)
      , cs_nextDocId       :: DocId
      , cs_readAttributes  :: Attributes      -- passed to readDocument
      , cs_tempPath        :: Maybe String
      , cs_crawlerTimeOut  :: Int
      , cs_fPreFilter      :: ArrowXml a' => a' XmlTree XmlTree
      , cs_fGetReferences  :: ArrowXml a' => a' XmlTree [URI]
      , cs_fCrawlFilter    :: (URI -> Bool) -- follow link or not
      , cs_fGetCustom      :: Custom a
      , cs_docs            :: HolDocuments d a => d a
      }
```

Listing 14: CrawlerState

on the website which must be crawled.

This leads to the assumption we could implement the process function as a monoid with an empty state as identity element and an `update` function for the state as binary operation.

Another, maybe more transparent solution, would be to extract the triplet from the current state and pass this triplet to the process function instead of the state (and also return a triplet). Listing 15 shows roghly an idea how the fold could be modified to achieve just this.

```
processCrawlResults oldCs _ l = do
  cs' <- updateStateWithTriplet (foldr process ( extractTriplet oldCs) l ) oldCs
  return $ Just cs'
```

Listing 15: Alternate reduce function for crawling

The function extractTriplet gets all the values needed for the fold with process and updateStateWithTriplet creates a new state from the generated triplet and the old state.

If now the calculation of each element of the triplet forms a monoid we can combine them with the predefined monoid of triplets as shown in listing 16.

The first element of the triplet would have a function which merges some sets of lists to create a new todo-list as binary operation (listing 17).

```
cs_toBeProcessed = S.union (cs_toBeProcessed cs)
                           (S.difference (S. filter (cs_fCrawlFilter cs) refs)
                                         (cs_wereProcessed cs))
```

Listing 17: Binary operation for the first element of the triplet

```haskell
instance (Monoid a, Monoid b, Monoid c) => Monoid (a,b,c) where
  mempty = (mempty,mempty,mempty)
  mappend (a1,b1,c1) (a2,b2,c2) =
      (mappend a1 a2, mappend b1 b2, mappend c1 c2)
```

Listing 16: Monoid of triplets

The binary operation of the second element is the insertion of the current document in a set of documents with **Nothing** as identity element to indicate there is no document to insert (listing 18).

```haskell
cs_docs = snd (insertDoc (cs_docs cs) (fromJust mdoc))
```

Listing 18: Binary operation for the second element of the triplet

The third element can be treated just as the second, as its binary operation basically is an insertion of a key/value pair in a Map (listing 19).

```haskell
cs_docHashes = if isJust (cs_docHashes cs)
                 then Just $ M.insert theMD5 (uri $ fromJust mdoc)
                        (fromJust $ cs_docHashes cs)
                 else Nothing
```

Listing 19: Binary operation for the third element of the triplet

These examinations lead to the conclusion that the reduce phase of the Holumbus Crawler can in fact be constructed in a monoidal manner, namely the monoid of triplets, composed of the three monoids we just outlined.

We will not discuss the second branch of the if clause in the process function as it is similar to the first branch.

2.5 Disco

Disco is an open-source implementation of the Map-Reduce framework for distributed computing. Like the original framework, Disco supports parallel computations over large data sets on unreliable cluster of computers. Like Hadoop, you can run Disco in the Amazon's Elastic Computing Cloud.

The Disco core is written in Erlang, a functional language that is designed for building robust fault-tolerant distributed applications. Users of Disco typically write jobs in Python but there is a possibility to use any language through an external interface. Disco provides an external interface for specifying map and reduce functions as external programs, instead of Python functions. It is also possible to make use of shared libraries using the `ctypes` foreign function interface of Python.

Disco was started at Nokia Research Center as a lightweight framework for rapid scripting of distributed data processing tasks. This far Disco has been succesfully used, for instance, in parsing and reformatting data, data clustering, probabilistic modelling, data mining, full-text indexing, and log analysis with hundreds of gigabytes of real-world data. [6]

Unfortunately the documentation does not provide many examples of MapReduce tasks, only the obligatory wordcount is explained in detail. However, from the previous description we can see what Disco (and thus MapReduce) is typically used for. We already have examined use cases for the mentioned categories: parsing and reformatting data (see the WordCount examples), probabilistic modelling (for example the Pi Estimator in section 2.2.3 and other MonteCarlo calculations), data mining (there are many examples of data mining with MapReduce, this is the most popular use case and not that different from parsing and reformatting. See section 2.1.6 for an example of summarizing the most important words of a document), full-text indexing (see the Inverted Index in section 2.1.7) and log analysis (for example server logs like counting the Most Popular URLs in section 2.3.2).

2.5.1 Wordcount

The map function for the WordCount example is very compact:

```
1 def map(e, params):
2         return [(w, 1) for w in e.split()]
```

Listing 20: Map function of a Disco WordCount

The first parameter contains an input entry, which is by default a line of input. The lines are split into words and every word is returned together with the value `1` as a tuple (of key and value). The second parameter params can be any object that you specify, in case that you need some additional input for your functions [1].

[6] `http://discoproject.org/`

```
def reduce(iter, out, params):
    stats = {}
    for word, count in iter:
        if word in stats:
            stats[word] += int(count)
        else:
            stats[word] = int(count)
    for word, total in stats.iteritems():
        out.add(word, total)
```

Listing 21: Reduce function of a Disco WordCount

The reduce function takes three parameters: The first parameter, iter, is an iterator that loops through the intermediate values produced by the map function, which belong to this reduce instance or partition.

In this case, different words are randomly assigned to different reduce instances. This is something that can be changed â see the parameter **partition** in disco.core.Job() for more information. However, as long as all occurrences of the same word go to the same $\mathcal{REDUCE}$ we can be sure that the final counts are correct.

So we iterate through all the words, and increment a counter in the dictionary stats for each word. Once the iterator has finished, we know the final counts, which are then sent to the output stream using the out object. The object contains a method, out.add(key, value), that takes a key-value pair and saves it to a result file.

The third parameter params can be any object that you specify, in case that you need some additional input for your functions [1].

We can see the Disco framework works similar to the Hadoop framework and thus uses the same monoid as the Hadoop WordCount.

2.6 Qt Concurrent

Qt Concurrent is a C++ template library for writing multi-threaded applications. Programs written with Qt Concurrent automaticallly adjust the number of threads used according to the number of processor cores available.

The library includes functional programming style APIs for for parallel list processing, a MapReduce implementation for shared-memory (non-distributed) systems, and classes for managing asynchronous computations in GUI applications. [7] MapReduce in Qt Concurrent is implemented to work on shared-memory systems, so instead of managing cluster nodes it manages threads on a single computer [8].

2.6.1 Wordcount

The reduce function of the Qt WordCount takes one intermediate result hash and aggregates it into the final result. Qt Concurrent will make sure only one thread calls this function at a time. This has two implications: there is no need to use a mutex lock when updating the result variable, and the system can be smarter about how it manages threads. If a thread tries to become the reducer thread while another thread is reducing, the first thread does not have to block but can put the result on the to-be-reduced queue and then call the map function on a new piece of data instead.

```
1 void reduce(WordCount &result, const WordCount &w)
2 {
3     QMapIterator<QString, int> i(w);
4     while (i.hasNext()) {
5         i.next();
6         result[i.key()] += i.value();
7     }
8 }
```

Listing 22: Reduce function for WordCount in Qt Concurrent

Again, this MapReduce implementation operates on a keyed domain, the result of the wordcount is stored in an array with the corresponding key.

[7] http://labs.trolltech.com/page/Projects/Threads/QtConcurrent

[8] http://labs.trolltech.com/blogs/2007/04/26/mapreduce-in-qt-concurrent/

2.7 Skynet

Skynet is an open source Ruby implementation of Google's MapReduce framework, created at Geni. Skynet is an adaptive, self-upgrading, fault-tolerant, and fully distributed system with no single point of failure. It uses a "peer recovery" system where workers watch out for each other. If a worker dies or fails for any reason, another worker will notice and pick up that task. Skynet also has no special âmasterâ servers, only workers which can act as a master for any task at any time. Even these master tasks can fail and will be picked up by other workers. [9]

2.7.1 Grep

Skynet comes with an example of a distributed grep which we already know from the Hadoop framework. Listing 23 shows a Ruby implementation of the reduce function.

```
1 # Takes an array that looks like  [ [word1, cnt], [word2,cnt], [word1, cnt] ]
2 def self.reduce(reduce_datas)
3   results = {}
4   reduce_datas.each do |reduce_data|
5     results[reduce_data[0]] ||= 0
6     results[reduce_data[0]] += reduce_data[1]
7   end
8   results
9 end
```

Listing 23: Grep reducer in Ruby

$\mathcal{REDUCE}$ takes an array of $\langle word, value \rangle$ pairs where `value` is either `1` or, if a partitioning step after $\mathcal{MAP}$ was involved, the intermediate value for the corresponding `key`. Then it looks for each tuple of the array if it contains a word and adds the `value` to a corresponding `results` mapping. `results` is also the return value of the reduce function. Like the grep example from the Hadoop section, it does not emit pairs of $\langle pattern, matchedstring \rangle$ as in Google's MapReduce paper but $\langle matchedstring, 1 \rangle$ pairs. Hence, we can use the same monoid as in section 2.2.2.

[9] `http://skynet.rubyforge.org/`

2.8 Sawzall

The Sawzall paper [20] gives some hints for more sophisticated uses of the MapReduce framework such as filtering, sampling, generating histograms and finding the most frequent items. For the implementation they use *aggregators*, a concept on top of the MapReduce framework. Aggregators are a specialization of MapReduce, specializing for performing various aggregations.

Generally speaking, in order to implement an application using the MapReduce model, the developer has to implement map and reduce functions (and possibly a combine function). However, a lot of applications related to counting and statistics computing have very similar characteristics. Aggregators abstract out the general patterns of these functions.

The Sawzall paper lists a number of typical aggregators.

2.8.1 Collection

The `Collection` Aggregator is a simple list of all the emitted values, including all duplicates, in arbitrary order. There is no need for a reduce phase and hence the identity reducer can be used.

2.8.2 Sample

`Sample` is like `Collection`, but chooses an unbiased sample of the emitted values. The size of the desired sample is provided as a parameter. The Sawzall paper does not mention any hints on how it is implemented. If we wanted to construct this aggregator as a monoid we could use a function that takes two lists of aggregated data and returns a list that is never larger than the desired sample size. If the two lists together are larger, they get resampled. That is, out of the two lists a set of values of *sample size* is generated randomly and passed as a list. This resample function would serve as our binary operation, the identity element would be the empty list.

2.8.3 Sum

The `Sum` Aggregator performs a summation of all the emitted values. The emitted values must be arithmetic values or compound values with arithmetic components which are summed elementwise. We already know the monoid over addition and, for compound values, the monoid of tuples which can hold several monoid instances.

2.8.4 Maximum

`Maximum` returns the highest-weighted values. The values are tagged with the weight, and the value with the highest weight is chosen. A parameter specifies the number of values to keep. The weight has its type provided in the declaration and

its value in the emit statement. This can be achieved in a monoidal way by letting $\mathcal{MAP}$ emit a list of $\langle value, weight \rangle$ pairs where `weight` is determined by a mapping of value to weight, for example by defining a constant for every value or a function like **length** value. These Maps are then reduced in a way that the binary operation for reduction always emits a Map with less or equal elements than the user has specified. If two Maps together have more than the allowed number of elements, the elements with the highest weight are chosen and emitted.

2.8.5 Quantile, Top and Unique

The `Quantile` aggregator uses the set of emitted values to construct a cumulative probability distribution represented by the quantile values for each increment of probability. The algorithm is a distributed variant of that of Greenwald and Khanna [13]. The Greenwald-Khanna algorithm is based on the idea that if a sorted subset $\{v_1, v_2, \dots\}$ of the input stream S (of current size n) can be maintained such that the ranks of v_i and v_{i+1} are within $2\varepsilon n$ of each other, then an arbitrary quantile query can be answered with precision εn [6].

`Top` estimates which values are the most popular. (By contrast, the `Maximum` aggregator finds the items with the highest weight, not the highest frequency of occurrence.) For large data sets, it can be prohibitively expensive to find the precise ordering of frequency of occurrence, but there are efficient estimators. The top table uses a distributed variant of the algorithm by Charikar, Chen, and Farach-Colton [7]. The algorithm is approximate: with high probability it returns approximately the correct top elements. Its commutativity and associativity are also approximate: changing the order that the input is processed can change the final results. To compensate, in addition to computing the counts of the elements, we compute the estimated error of those counts. If the error is small compared to the counts, the quality of the results is high, while if the error is relatively large, chances are the results are bad.

This ”approximate” associativity is not enough for defining a monoid. A different evaluation order can yield a different result. The assumption here is that the data is suitable for the algorithm and therefore the output is a good approximation of the exact result. However, the Sawzall paper also states both, `Quantile` and `Top` are associative: ”For more sophisticated aggregators, such as `quantile` and `top`, care must be taken to choose an algorithm that is commutative, associative and reasonably efficient for distributed processing.” A deeper analysis of these algorithms to inspect their associativity and possibly form a monoid is out of scope of this paper.

The Unique aggregator reports the estimated size of the population of unique items emitted to it. In contrast to a Sum aggregator, Unique ignores duplicates and, for the sake of efficiency, uses a statistical algorithm that only generates an approximation of the exact value, just like the Quantile and Top aggregators.

2.9 Hadoop and aggregators

The Hadoop framework also offers the possibility of using a programming style based on aggregators. A wordcount serves as example of an implementation.

2.9.1 AggregateWordCount

The package provides generic mapper/reducer/combiner classes, a set of built-in value aggregators and a generic utility class that helps users to create map/reduce jobs using the generic class. The aggregator used in this example is called *sum over numeric values*.

The developer using aggregators will only need to provide a plugin class conforming to a ValueAggregatorDescriptor interface (see listing 2.9.1).

```
1 public interface ValueAggregatorDescriptor {
2   public static final String TYPE_SEPARATOR = ":";
3   public static final Text ONE = new Text("1");
4   public ArrayList<Entry<Text, Text>> generateKeyValPairs(Object key,
5                                                           Object val);
6   public void configure(JobConf job);
7 }
```

Its main function is to generate a list of aggregation-id/value pairs. An aggregation id encodes an aggregation type which is used to guide the way to aggregate the value in the reduce phase of an aggregator based job. The mapper in an aggregator based MapReduce job may create one or more ValueAggregatorDescriptor objects at configuration time. For each input key/value pair, the mapper will use those objects to create aggregation id/value pairs.

```
1 public static class WordCountPlugInClass
2               extends ValueAggregatorBaseDescriptor {
3   @Override
4   public ArrayList<Entry<Text, Text>> generateKeyValPairs(Object key,
5                                                           Object val) {
6     String countType = LONG_VALUE_SUM;
7     ArrayList<Entry<Text, Text>> retv = new ArrayList<Entry<Text, Text>>();
8     String line = val.toString ();
9     StringTokenizer itr = new StringTokenizer(line);
10    while ( itr .hasMoreTokens()) {
11      Entry<Text, Text> e = generateEntry(countType, itr.nextToken(), ONE);
12      if (e != null) retv.add(e);
13    }
14    return retv;
15  }
16 }
```

The package also provides a base class, ValueAggregatorBaseDescriptor, implementing the above interface. The user can extend the base class and implement the function generateKeyValPairs accordingly. In the case of the wordcount example the user has to provide a plugin-class like WordCountPlugInClass (shown in listing 2.9.1).

It reads the text input files, breaks each line into words and counts them. The output is a locally sorted list of words and the count of how often they occurred. To actually start the aggregation, the plugin-class is invoked by the ValueAggregatorJob class which creates the MapReduce job and handles the configuration for the job.

2.10 Comparison Of MapReduce Frameworks

All frameworks we looked at have quite different purposes and characteristics. Hadoop is probably the most sophisticated and also most popular open source framework while the Real World Haskell example is only that — an example of distributed programming in Haskell with no ambitions to become a widely used and fully fledged MapReduce implementation. Holumbus is a more complete framework also written in Haskell with support for a distributed file system.

Table 1 shows some differences of the frameworks we used. The implementation languages cover a wide range from functional language like Erlang and Haskell to the more classic imperative languages like C++ or Java. The original purpose of MapReduce is to run it on a large cluster of distributed machines, but not all frameworks provide this capability. QtConcurrent can only be used in a multicore environment but not in a distributed way. The same is true for the Real World Haskell example as already mentioned. Then, there is the possibility to use Sawzall-like aggregators to further abstract the MapReduce idea. Of the examined frameworks, only Hadoop supports this style of programming MapReduce tasks. The next column shows how the output is handled. `parameter` indicates that an `output` argument is passed to the reduce function and modified. This is the more imperative way to deal with the results while passing a return value is considered a more functional style. Finally, some frameworks allow the map and reduce functions to be written in other languages than the implementation language. Hadoop Streaming is a utility which allows Hadoop users to create and run jobs with any executables (e.g. shell utilities) as the mapper and/or the reducer. Hadoop Pipes is a SWIG-compatible C++ API to implement Map/Reduce applications [2]. Typically, Disco MapReduce jobs are written in Python, but Disco also provides an external interface for specifying map and reduce functions as external programs, instead of Python functions [10].

[10] http://discoproject.org/doc/external.html

Table 1: Characteristics of examined frameworks

	implementation language	distributed	aggregators	handling of output	other languages encouraged
Disco	Erlang	yes	no	parameter	yes
Hadoop	Java	yes	yes	parameter	yes
Holumbus	Haskell	yes	no	return value	no
MapReduce (Google)	C++	yes	no	return value	no
QtConcurrent	C++	no	no	parameter	no
Real World Haskell	Haskell	no	no	return value	no
Sawzall	C++	yes	yes	return value	no
Skynet	Ruby	yes	no	return value	no

2.11 Further examples of monoids

Of course there are many more monoids than discovered in the previous examples. Here is an incomplete list of possible monoids which are also included in the Haskell module `Data.Monoid`[11]:

Max The tuple `(max, minBound)` forms a monoid for every data structure of type `Ordering`. `Ordering` is a Haskell primitive with the values `LT, EQ` and `GT` resembling *lesser than*, *equal* and *greater than*.

Min Same as `Max`, but with `min` as binary operation and `maxBound` as identity element.

MaxPriority and MinPriority The monoid `(max, Nothing)` and its counterpart `(min, Nothing)` are similar to `Max` and `Min` with the difference of the domain which is `Maybe a` where a is of type `Ordering`. Nothing is the identity element (representing the bottom element for `MaxPriority` and the top element for `MinPriority`.

All This is the boolean monoid under conjunction `(Bool, &&, True)` which fails if any value is `False`.

Any This is the boolean monoid under disjunction `(Bool, ||, False)` which succeeds if any value is `True`.

Sum We already know the monoid under addition `(+, 0)` from several MapReduce examples. We restricted the domain to Integer values, but the monoid can be used with every domain in which addition is defined. In Haskell terms we would say it is restricted to the `Num` type class.

Product This is the monoid under multiplication `(*, 1)`.

First and Last Maybe monoids returning the leftmost non-`Nothing` value and the rightmost non-`Nothing` value. For example First (**Just** 'a') 'mappend' First (**Just** 'b') results in **Just** 'a'. `Nothing` is the identity element.

Dual The dual of a monoid, obtained by swapping the arguments of mappend. For non-commutative monoids (like `First` or `Last`) the result changes. For example Dual (First (**Just** 'a')) 'mappend' Dual (First (**Just** 'b')) will have **Just** 'b' as result whereas First (**Just** 'a') 'mappend' First (**Just** 'b') will result in **Just** 'a'.[14]

[11] http://hackage.haskell.org/packages/archive/base/4.1.0.0/doc/html/src/Data-Monoid.html

List monoid The list monoid as described in the `grep` example in section 2.1.2.

Maybe This is just a wrapper if we encounter a monoid encased in the `Maybe` type: **Just** (Sum 3) ‘mappend‘ **Just** (Sum 4) results in **Just** 7 as expected.

Endo The monoid of endomorphisms under composition. Endomorphism is such morphism (morphism is another term for homomorphism) whose source and destination are the same object. That is a morphism f is endomorphism, when $\mathrm{Src} f = \mathrm{Dst} f = A$ where A is some object (e.g. A may be an abstract algebra). Then one can say, the object of endomorphism f is A. [12]

We can say function composition forms a monoid if the functions are endomorphisms.

Tuples A tuple consisting of monoids is a monoid itself. The binary operation for tuples is defined by applying binary operations in a component-wise manner, and the unit is just the tuple of units for the component types[17].

[12] http://planetmath.org/encyclopedia/Automorphism6.html

Table 2: A list of monoids from the discussed use cases

	Map			Integer	Special algorithms					Composition	
	addition	concatenation	unionWith (+)	addition	identity	resample*	merge	GK*	CCF*	triplet	any tuples
WordCount	1, 2, 4, 5			3							
Sort					2, 3, 4						
Grep	2, 8	4			3						
PiEstimator	2										
Sudoku					2						
Pentomino					2						
Counting Lines				6							
URL Access Frequency	4										
Reverse Web-Link Graph		4									
Inverted Index		4									
Most Popular URLs	6										
Term-Vector per Host			4								
Crawler										3	
Collection aggregator					8						
Sample aggregator						8					
Sum aggregator				8							8
Maximum aggregator							8				
Quantile aggregator								8			
Top aggregator									8		
Unique aggregator											

1: Disco framework
2: Hadoop framework
3: Holumbus framework
4: MapReduce (Google)
5: QtConcurrent
6: Real World Haskell
7: Sawzall
8: Skynet
*: not a monoid

3 Implementation

3.1 Built-in monoids

Haskell already comes with predefined monoids aswell as a possibility to implement monoids ourselves. The following code examples are taken mostly from Edward Kmett and his work about efficient reducers using monoids [15]. The Haskell module Data.Monoid defines the Monoid type class as shown in listing 24. mempty defines the identity element, mappend the associative binary operation.

```
1 class Monoid m where
2 mempty      :: m
3 mappend     :: m -> m -> m
```

Listing 24: The monoid type class (1)

mconcat (see listing 25) is a convenience function that is not necessary to define the monoid. It just applies mappend to a list of monoid values.

```
1 mconcat     :: [m] -> m
2 mconcat = foldr mappend mempty
```

Listing 25: The monoid type class (2)

As described in section 2.11 there is a number of built-in monoids. Here we examine the implementation of some of them.

Monoid under addition To define a concrete monoid we create an **instance** of the Monoid type class. This **instance** then defines the two functions mempty and mappend. We also define the type of the monoid as Sum a where a can be a type of the domain of the monoid. In this case we restrict a to the **Num** type class which also covers the classes **Real**, **Fractional**, **Integral**, **RealFrac**, **Floating** and **RealFloat** as seen in figure 5. This means we can pass any number to this monoid where addition is defined.

We can now specify mempty and mappend. mappend is the equivalent of addition, so mappending two Sums should yield the same result as the ordinary addition, just wrapped in the Monoid type. mempty, representing the identity element, is 0 wrapped in the Monoid type.

```
1 newtype Sum a = Sum a
2 instance Num a => Monoid (Sum a) where
3   mempty = Sum 0
4   Sum a `mappend` Sum b = Sum (a + b)
```

Listing 26: The monoid under addition

We can test the implementation: the outcome of Sum 3 ‘mappend‘ Sum 4 is Sum 7 and Sum 3 ‘mappend‘ mempty results in Sum 3. The monoid under multiplication is defined in the same manner.

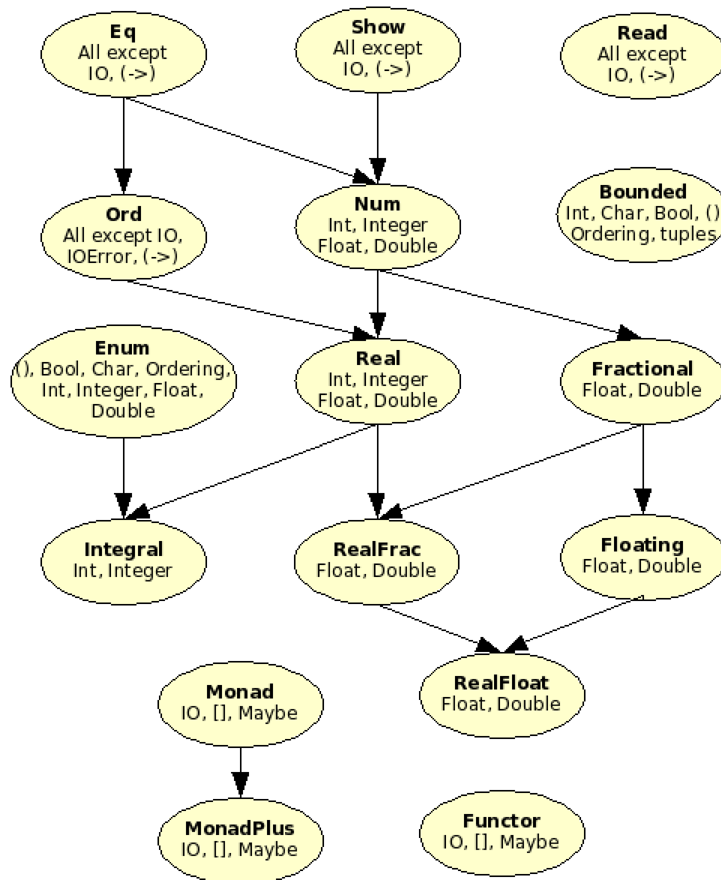


Figure 5: Haskell standard type classes

Monoid under concatenation The monoid under concatenation is similar to the monoid under addition. Here we do not need to define a new type, instead the built-in list type, [], is used. Another difference is the lack of a type restriction in the definition of the monoid. We can use this monoid with values of arbitrary types. The identity element is the empty list and the binary operation is the concatenation of two lists as described in section 2.2.2.

```
1 instance Monoid [a] where
2   mempty = []
3   mappend = (++)
```

Listing 27: The monoid under concatenation

Max monoid The Max monoid is a monoidal way to calculate the maximum of a number of values. **minBound** represents the identity element and the Haskell function **max** is the binary operation.

```
newtype Max a = Max { getMax :: a } deriving (Eq,Ord,Show,Read,Bounded)
instance (Ord a, Bounded a) => Monoid (Max a) where
  mempty = Max minBound
  mappend = max
```

Listing 28: The Max monoid

Monoid unter composition Another example of a built-in monoid is the monoid unter composition, the Endo monoid. mappend is function composition, mempty the identity function. We can use the monoid with the appEndo function, which applies the endomorphisms to an initial value: appEndo (Endo (+2) ‘mappend‘ Endo (*3)) 4 results in 14 as $4 * 3 + 2 = 14$.

```
newtype Endo a => Endo (a -> a)
instance Monoid (Endo a) where
  mempty = id
  Endo f ‘mappend‘ Endo g = Endo (f . g)
```

Listing 29: The monoid under composition

Composition of monoids We can combine two (or more) monoids into one monoid of tuples. The binary operation for tuples is defined by applying binary operations in a component-wise manner, and the unit is just the tuple of units for the component types [17]. Listing 30 shows the composition of two monoids into a doublet of monoids.

```
instance (Monoid m, Monoid n) => Monoid (m,n) where
  mempty = (mempty,mempty)
  (a,b) ‘mappend‘ (c,d) = (a ‘mappend‘ c, b ‘mappend‘ d)
```

Listing 30: Monoid of tuples

3.2 Non-standard monoids

In sections 2.2.4,2.1.6 and 2.3.2 we could not use predefined monoids but defined our own definitions.

AscendingSet This is the monoid used for sorting in section 2.2.4. The main data structure is the AscendingSet which is a set of already sorted values. The monoid‘s mappend operation is essentially a union operation on (ordered) sets; it performs a merge step (in the sense of merge sort) on the opaque list representation [17].

```
newtype Ord x => AscendingSet x = AscendingSet { getAscendingSet :: [x] } deriving (Show)
instance Ord x => Monoid (AscendingSet x) where
  mempty = AscendingSet []
```

```
4   mappend x y = AscendingSet $ merge compare (getAscendingSet x) (getAscendingSet y)
```

Listing 31: Monoid of ascending sets

```
1 merge c   []    y    =  y
2 merge c   x     []   =  x
3 merge c   xs    ys   =  case c (head xs) (head ys) of
4                           EQ -> (head xs)  :  merge c (tail xs) (tail ys)
5                           LT -> (head xs)  :  merge c (tail xs) ys
6                           GT -> (head ys)  :  merge c xs (tail ys)
```

Listing 32: Merge function for ascending sets

3.2.1 Monoids over Maps

Monoid under addition Many examples required a monoid over Maps, merging them by key and adding the corresponding values. The unionWith function in the Data.Map module provides exactly this behaviour. Listing 36 shows the implementation of mappend with unionWith.

```
1 data Map' k v = Map' { getMap' :: Map k v } deriving (Show)
2 instance (Ord k, Ord v, Num v) => Monoid (Map' k v) where
3   mempty = Map' $ fromList []
4   mappend x y = Map' $ unionWith (+) (getMap' x) (getMap' y)
```

Listing 33: A monoid over Maps

We need to specify the type Map' because there already is an implementation of a monoid over Maps in Haskell which does not offer the functionality we need. Listing 34 shows the default definition of the Map monoid.

```
1 instance (Ord k) => Monoid (Map k v) where
2     mempty = empty
3     mappend = union
4     mconcat = unions
```

Listing 34: The built-in monoid over Maps

Monoid under concatenation This monoid is almost the same as the previous, differing only in the definition of mappend.

```
1 data Map' k v = Map' { getMap' :: Map k v } deriving (Show)
2 instance (Ord k, Ord v, Num v) => Monoid (Map' k v) where
3   mempty = Map' $ fromList []
4   mappend x y = Map' $ unionWith (++) (getMap' x) (getMap' y)
```

Listing 35: A monoid over Maps

Monoid with Nested addition This monoid is only used in section 2.1.6 in the term vector example. Basically the values that we want to add are wrapped in yet another Map so we have to add up the inner Maps first and then aggregate the results.

```
data Map' k v = Map' { getMap' :: Map k v } deriving (Show)
instance (Ord k, Ord v, Num v) => Monoid (Map' k v) where
  mempty = Map' $ fromList []
  mappend x y = merge

merge x y = Map' $ unionWith (unionWith (+)) (getMap' x) (getMap' y)
```

Listing 36: A monoid over Maps

Maximum aggregator The Sawzall Maximum aggregator searches for the keys with the highest values where the number of results can be specified by the user (in the example in listing 37 it calculates the top six values). For reasons of efficiency the result of the merge function is always a Map of less elements than specified by the user.

```
data Map' k v = Map' { getMap' :: Map k v } deriving (Show)

instance (Ord k, Ord v, Num v) => Monoid (Map' k v) where
  mempty = Map' $ fromList []
  mappend x y = Map' $ merge (getMap' x) (getMap' y) 6

merge :: (Ord k, Ord v, Num v) => Map k v -> Map k v -> Int -> Map k v
merge x y n
  | size myUnion <= n = myUnion
  | otherwise = takeSomeOf myUnion
  where
    myUnion = unionWith (+) x y
    f = flip (comparing snd)
    takeSomeOf = fromList . take n . sortBy f . toList
```

Listing 37: The Maximum monoid

3.3 Non-Trivial Monoids/Reducers

Edward Kmett mentioned some more advanced topics where monoids can be utilized [15]. These topics cannot be examined in this paper but may give an idea of further monoid driven use cases:

- Tracking Accumulated File Position Info
- FingerTree Concatenation
- Delimiting Words

- Parsing UTF8 Bytes into Chars
- Parsing Regular Expressions
- Recognizing Haskell Layout
- Parsing attributed PEG, CFG, and TAG Grammars

4 Conclusion

Google's MapReduce programming model has become popular in the context of parallel, distributed computing and has lead to several frameworks which adopted their idea. Some projects, and Google themselves with their domain specific language Sawzall, extended the original MapReduce model with the concept of aggregators, an approach based on monoidal abstractions. We showed that most use cases can be modeled in a strict monoidal way with the notably exceptions of the statistical aggregators *Top*, *Quantile* and *Unique* from the Sawzall paper. They break the associativity of the underlying binary operation which is a necessary prerequisite to construct a monoid. It turned out that an *approximate* associativity is sufficient for large statistical aggregations when the resulting error is acceptable.

References

[1] Disco documentation, http://discoproject.org/doc/start/tutorial.html.

[2] Hadoop documentation, http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/examples/dancing/package-summary.html.

[3] The holumbus framework - creating scalable and highly customized crawlers and indexers.

[4] Holumbus mapreduce examples, http://holumbus.fh-wedel.de/trac/wiki/MapReduceExamples.

[5] Solving pentomino puzzles with backtracking, http://www.yucs.org/~gnivasch/pentomino/.

[6] Chiranjeeb Buragohain and Subhash Suri. Quantiles on streams. In *Encyclopedia of Database Systems*, pages 2235–2240. 2009.

[7] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. pages 693–703, 2002.

[8] Hung chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, New York, NY, USA, 2007. ACM.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. pages 137–150, 2004.

[10] Katrin Erk and Lutz Priese. *Theoretische Informatik*. eXamen.press. Springer Verlag, 3 edition, 2008. 485 Seiten, 115 Abbildungen.

[11] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, January 1979.

[12] Dan Gillick, Arlo Faria, and John Denero. Mapreduce: Distributed computing for machine learning, 2006.

[13] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD Conference*, pages 58–66, 2001.

[14] Mark P. Jones. Functional programming with overloading and higher-order polymorphism, 1995.

[15] Edward Kmett. All about monoids, `http://comonad.com/reader/2009/hac-phi-slides/`.

[16] Donald E. Knuth. Dancing links, 2000.

[17] Ralf Lämmel. Google's mapreduce programming model — revisited. *Sci. Comput. Program.*, 68(3):208–237, 2007.

[18] Andy Oram and Greg Wilson. *Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice (O'Reilly))*. O'Reilly Media, Inc., June 2007.

[19] Bryan O'Sullivan, Donald Stewart, and John Goerzen. *Real World Haskell*. O'Reilly Media, Inc., December 2008.

[20] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 14, September 2006. Special Issue: Dynamic Grids and Worldwide Computing.

[21] Christian P. Robert and George Casella. *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer, July 2005.

[22] David E. Rydeheard and Rod M. Burstall. *Computational category theory*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1988.