Bloc: Library for handling large binary objects in Haskell

A thesis submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Anshu Avinash

to the
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
June, 2015

CERTIFICATE

It is certified that the work contained in the thesis titled **Bloc: Library for handling large binary objects in Haskell**, by **Anshu Avinash**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Piyush Kurur

Department of Computer Science & Engineering

IIT Kanpur

June, 2015

ABSTRACT

Name of student: **Anshu Avinash** Roll no: **10327122**

Degree for which submitted: **Master of Technology**

Department: **Computer Science & Engineering**

Thesis title: **Bloc: Library for handling large binary objects in Haskell**

Name of Thesis Supervisor: **Piyush Kurur**

Month and year of thesis submission: **June, 2015**

In this thesis, we describe a library for handling large binary objects (blob) written in Haskell - a purely-functional programming language. We use the idea of storing each blob as a separate file. We also try to make all the operations on a blob to be safe under concurrent access without using any locks. We leverage many features offered by Haskell like modularity and strong type system.

Dedicated to the Haskell community

Acknowledgements

This thesis would have been impossible without the guidance from my thesis advisor, Piyush Kurur. He came up with the topic of this thesis, taking into consideration my interest in Functional programming and systems. He was always accessible for discussions, sometimes not necessarily related to the thesis. I would like to express my sincere gratitude towards him.

I would also like to thank the Haskell community (#haskell irc channel, /r/haskell on reddit) for making the journey of understanding Haskell pleasant.

I am thankful to the Department of Computer Science and Engineering, IIT Kanpur, for providing the necessary infrastructure for my research work.

My family and friends always supported me throughout my thesis work - special thanks to them.

Contents

List of Tables

List of Figures

List of Programs

Chapter 1

Introduction

Most of the web applications today require to store some kind of data persistently. For a web application that handles student management - the data can be name, date of birth, photograph and other information about students.

These web applications use one of the *databases* to store their data. Database refers to a collection of information that exists overs a long period and a Database Management System (DBMS) is a tool for creating and managing large amount of data efficiently.

Early database management systems evolved from file systems. These database systems used tree-based and the graph-based models for describing the structure of the information in a database. Database systems changed significantly following a famous paper written by Ted Codd in 1970 [1]. Codd proposed that database systems should present the user with a view of data organized as tables called relations. This paper was foundation for popular relational databases like MySQL and PostgreSQL.

However, many of the web applications do not require the complex querying and management functionality offered by a Relational Database Management System (RDBMS). This among other reasons gave rise to NoSQL (Not only SQL) databases. These databases can be classified based on the data models used by them. Amazon's Dynamo [2] is a key-value store. In key-value stores records are stored and retrieved using a key that uniquely identifies the record. MongoDB [3] on the other hand is a document-oriented database. Document-oriented databases are designed for

managing document-oriented information, also known as semi-structured data.

Today's web applications also work with large files like images, music, videos etc. Size of these files can vary from few MBs to tens of GBs. The application developer can decide to store these files directly into the one of the databases mentioned above or store it as a file and save the filename in the database. This large binary data is usually called a blob (**B**inary **L**arge **O**bject).

In this thesis, we provide a simple interface written in *Haskell* for handling blobs. We provide methods for incremental writing, incremental reading and garbage collection of deleted blobs.

Another important property of web applications is concurrency. Concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other. Concurrency plays an important part in building scalable and fault tolerant web applications. However, building concurrent systems usually requires working with locks and may result into issues like deadlock and starvation.

We provide an alternative to using locks at application level for concurrency. We make use of the atomic guarantees provided by the operating system on certain file operations to provide concurrency for our implementation.

1.1 Organization of the thesis

Chapter 2 discusses the approach of storing large files in databases. It also provides a background for this thesis work. In Chapter 3, we give a brief introduction to functional programming. Chapter 4 describes our design and implementation. We conclude and present the future work in Chapter 5.

Chapter 2

Related Work and Background

2.1 Storing large objects

Large object files can either be stored directly in a database or we can store the path to the binary file and other metadata. In this section we will discuss few examples of both. We will also discuss merits and demerits of both the approaches.

2.1.1 Storing large objects in database

Exodus was one of the first databases to support storage of large object files [4]. It used B+ tree index on byte position within the object plus a collection of leaf (data) blocks. Exodus allowed searching for a range of bytes, inserting a sequence of bytes at a given point in the object, appending a sequence of bytes at the end of the object and to delete a sequence of bytes from a given point in the object.

Popular relational databases like MySQL and PostgreSQL both provide data types to store large object files. In MySQL the data type is called BLOB, and has operations similar to that on a string. Corresponding data type in PostgreSQL is bytea.

PostgreSQL also provides a BLOB data type which is quite different from MySQL's BLOB data type. It's implementation breaks large objects up into "chunks" and stores the chunks in rows in the database. A B-tree index guarantees fast searches

for the correct chunk number when doing random access reads and writes.

A similar idea is used by MongoDB, which is a document database. It also divides the large object into "chunks". It uses GridFS specification for this [5]. GridFS works by storing the information about the file (called metadata) in the "files" collection. The data itself is broken down into pieces called chunks that are stored in the "chunks" collection.

2.1.2 Storing metadata and filename in database

Another approach to store large objects is to store only the filename and some metadata in the database. In this case the application has to take care of the all externally attached files as well as the security settings.

2.1.3 Comparison of both approaches

Both the approaches have their own benefits and disadvantages.

- **Performance**

  When we just store the filename in database, we skip the database layer altogether during file read and write operations. In the paper To BLOB or Not To BLOB [6], performance of SQL Server and NTFS has been compared. The results showed that the database gave higher throughputs for objects for relatively small size ($<$ 1MB).

- **Security**

  Security and access controls are simplified when the data is directly stored in the database. When accessing the files directly, security settings between file system and database are independent from each other.

- **ACID guarantees**

  ACID stands for Atomicity, Consistency, Isolation and Durability. All RDBMS give ACID guarantee on database transactions. If the large object is directly

stored in a relational database, all the operations on it also offer ACID guarantees. If the large object is stored as a separate file, you don't get any such guarantee. Also, note that this point is not valid for NoSQL databases.

2.2 Concurrency

Concurrency is a program-structuring technique in which there are multiple *threads of control*. Concurrency is concerned with nondeterministic composition of programs (or their components).

In many of the applications today, concurrency is a necessity. For example, a server-side application needs concurrency in order to manage multiple client interactions simultaneously.

2.2.1 Concurrent Programming Models

There are multiple concurrent programming models like: actors, shared memory and transactions. The traditional way of offering concurrency in a programming language is by using threads which operate on a shared memory. In this model, *locks* are used to achieve mutual exclusion. However, there are several problems with using locks [7]:

- Taking too few locks - One might forget to take a lock, resulting into two variables modifying same variable simultaneously.
- Taking too many locks - Taking too many locks might result into a deadlock or avoid concurrency altogether.
- Taking locks in the wrong order - Acquiring locks in the wrong order can result into a deadlock.
- Error recovery - It is difficult to guarantee that no error can leave the system in an inconsistent state.

The other two approaches try to avoid this problem of using locks. Actor model treats "actors" as the primitives of concurrent computation [8]. It provides an approach to concurrency that is entirely based on passing messages between processes. In Erlang, which is used to build scalable real-time systems, actors are part of language itself [9].

Software Transactional Memory (STM) is a technique for simplifying concurrent programming by allowing multiple state-changing operations to be grouped together and performed as a single atomic operation.

Haskell supports all the above three concurrent programming models. However, in this thesis we try to achieve concurrency by having a design which uses "atomic" file operations provided by the operating system.

2.2.2 Atomicity of file operations

In this section, we talk about atomicity of operations on file systems which provide a POSIX-file system based view.

We use the `rename` system call [10] for moving a file to a new destination. `rename` is atomic in the sense that if it is used to overwrite a file, it is atomically replaced, so that there is no point at which another process attempting to access the file will find it missing. Also, at any time, a process will not be able to read any "partial" file. This atomic property of `rename` is used by *maildir* [11] to maintain message integrity as messages are added.

Another system call `mkdir`, which is used to create a new directory, is also atomic [12]. So, if two processes try to create the same directory simultaneously, only one of them will succeed and other will receive the "EEXIST: path name already exists" error.

Chapter 3

Functional Programming

In functional programming the fundamental operation is the application of functions to arguments. The main program itself is written as a function that receives the program's input as its arguments and delivers the program's output as its result [13]. In this section we will discuss special characteristics and advantages of functional programming (Haskell in particular).

3.1 Referential Transparency

Functional programs contain no assignment statements, so variables, once given a value, never change. More generally, functional programs contain no side-effects at all. A functional call can have no effect other than to compute its result. This eliminates a major source of bugs, and also makes the order of execution irrelevant - since no side effect can change an expression's value, it can be evaluated at any time. Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa - that is, programs are "referentially transparent".

3.2 Statically typed

Every expression in Haskell has a type which is determined at compile time. All the types composed together by function application should match up. If they don't,

the program will be rejected by the compiler.

3.3 Algebraic Data Types and Pattern Matching

An algebraic data type has one or more data constructors, and each data constructor can have zero or more arguments. These algebraic data types can be recursive too. We can define functions on algebraic data types using pattern matching. In pattern matching, we attempt to match values against patterns and, if so desired, bind variables to successful matches.

Program 3.1 Pattern matching on algebraic data types

```
data Shape = Rectangle Int Int
           | Square Int

area :: Shape -> Int
area (Rectangle len breadth) = len * breadth
area (Square side)           = side * side

rec = Rectangle 3 4

main = print $ area rec
```

3.4 Lazy Evaluation

In lazy evaluation, an expression is not evaluated until its value is needed. This implies that programs can compose very well. Laziness also allows us to construct infinite data structures. Consider this example of generating primes.

Program 3.2 Program to generate list of primes

```
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]

main = print $ take 10 primes
```

The `primes` method generates an infinite list of primes lazily. In the main method, we take first 10 primes from the list and print them. On running, the above program generates the correct output: `[2,3,5,7,11,13,17,19,23,29]`.

Note that we did not specify the type of `primes` and `main`. In Haskell, we don't have to explicitly write out every type. Types are inferred by unifying every type bidirectionally.

3.5 Foreign Function Interface (FFI)

The Foreign Function Interface allows Haskell programs to cooperate with programs written in other languages. Haskell FFI is very easy to use. Consider an example of calling the power function ("pow") of the libc.

Program 3.3 Calling C's pow function from Haskell

```
foreign import ccall "pow" c_pow :: Double -> Double -> Double

main = print $ c_pow 3.0 4.0
```

When executed, this program calls the pow function of libc and prints the result 81.0.

Chapter 4

Design and Implementation

Our design is inspired from the maildir format [11]. The maildir format stores each message in a separate file with a unique name. The mail user agent (MUA) does not have to worry about partially delivered mail: each message is safely written to disk in the *tmp* subdirectory before it is moved to *new*. When a mail user agent process finds message in the *new* subdirectory, it moves them to *curr*.

Similar to maildir, we also store all large objects in separate files. All the large objects of a database are stored under a single directory which we also call a "BlobStore". The BlobStore contains three subdirectories: *tmp*, *curr* and *gc*. We will discuss purpose of these directories later in this chapter.

4.1 Initializing the BlobStore

Before starting to create blobs inside a directory, we ensure that the *tmp* and *curr* subdirectories have already been created. We provide a method `openBlobStore` which takes the path of a directory which is to be used as BlobStore as argument and does the initialization for us. BlobStore is defined as a newtype. We don't expose the constructor for BlobStore, so the only way to get a BlobStore is by using the `openBlobStore` method.

Program 4.1 Definition of BlobStore

```
newtype BlobStore = BlobStore FilePath
```

4.2 Creating a new Blob

We provide a method called `newBlob` for creating a new blob. It takes a BlobStore as a parameter and returns a WriteContext.

Program 4.2 Definition of WriteContext

```
data TempLocation = TempLocation FilePath FilePath

data WriteContext = WriteContext { writeLoc    :: TempLocation
                                 , writeHandle :: Handle
                                 , hashCtx     :: Ctx
                                 }
```

WriteContext contains the file handle of just created blob, a TempLocation and a hashCtx. TempLocation stores the base directory and the filename of just created blob. The hashCtx is used to store the SHA-512 hash of the contents that has been written to the blob.

All the new blobs are created in the *tmp* folder. We use Version 4 UUID [14] to give unique names to the newly created blobs.

4.3 Writing to a Blob

Since the blobs can be very large, it is possible that the entire blob will not fit in the main memory. Hence, we provide an incremental interface for writing to a blob.

We only allow to add new data at the end of a given blob. `writePartial` takes a blob and a WriteContext as arguments and appends the given blob to the WriteContext's blob.

Once all the data has been written to the blob, `endWrite` is called. `endWrite` takes a WriteContext as argument and moves the blob from *tmp* folder to *curr* folder.

Before moving the blob, we ensure that all the data has been written to disk by calling the `fsync` system call.

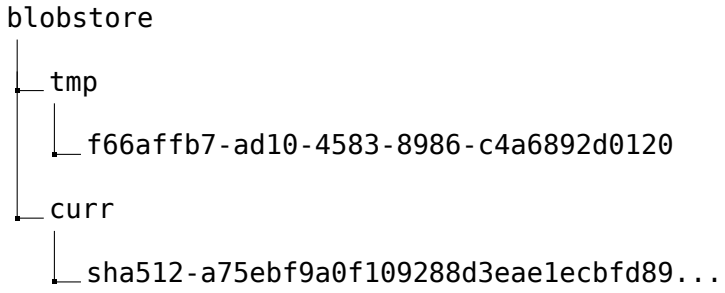
In `endWrite`, we also rename the file to SHA-512 hash of its contents. Using hash of the contents as filename ensures that if multiple blobs have same contents, only one copy is stored. We also prefix the filename with the name of the hash function i.e. "sha-512". This is for future compatibility, in case we want to support other hash functions.

`endWrite` returns a BlobId. This BlobId contains the location of the blob. No more updates to the blob are possible after calling `endWrite`.

Program 4.3 Definition of BlobId

```
data BlobId = BlobId FilePath FilePath
```

Figure 4.1: Directory structure of a BlobStore



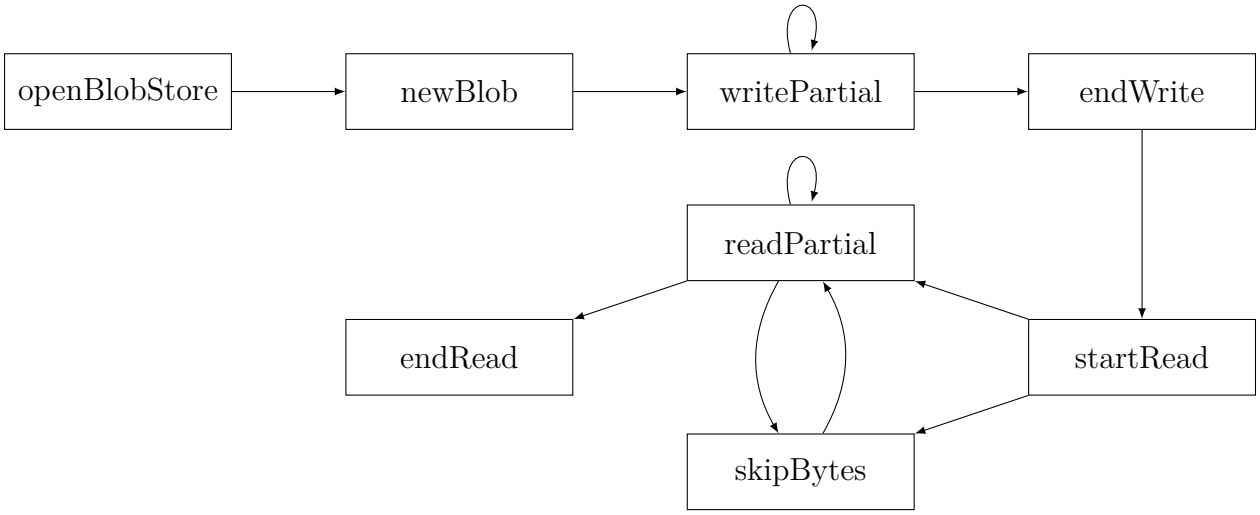
4.4 Reading from Blob

Similar to writes, reading is also incremental. First the `startRead` method is called which takes a BlobId as argument and returns a ReadContext. ReadContext contains the file handle of the blob which is opened in read mode.

Program 4.4 Definition of ReadContext

```
data ReadContext = ReadContext Handle
```

Figure 4.2: Order of operations on Blob



`readPartial` takes a ReadContext and number of bytes as input and returns those number of bytes from the blob. While reading from a blob, you can skip ahead using the method `skipBytes`. `skipBytes` takes a ReadContext and number of bytes, b as input and skips b bytes ahead in the ReadContext. The `endRead` method closes the open handle.

Table 4.1: Interface for operations on blob

Method	Purpose
`openBlobStore`	Initializes given directory to be used as a BlobStore
`newBlob`	Creates a blob in the given BlobStore
`writePartial`	Takes a blob and appends it to the end of the blob given in the argument
`endWrite`	Takes a WriteContext as input and returns a BlobId
`startRead`	Takes a BlobId as input and returns a ReadContext
`readPartial`	Reads a given number of bytes from a Blob
`skipBytes`	Skips ahead a given number of bytes in a Blob
`endRead`	Completes the read

4.5 Helper methods for small blobs

Blobs which can easily fit into the main memory, do not require the incremental write and read methods. For such blobs, we provide helper methods `createBlob`

and `readBlob`.

`createBlob` takes a BlobStore and the blob contents as arguments and returns the BlobId of the newly created blob. It basically combines the newBlob, writePartial and endWrite methods.

Program 4.5 Definition of createBlob

```
createBlob :: BlobStore -> Blob -> IO BlobId
createBlob blobstore blob = newBlob blobstore
  >>= \wc -> writePartial wc blob
  >>= endWrite
```

`readBlob` takes a BlobId as argument and returns its entire contents.

Table 4.2: Helper methods for small blobs

Method	Purpose
`createBlob`	Creates a blob with a given content in a BlobStore
`readBlob`	Takes a BlobId as argument and reads its entire contents

4.6 Garbage Collection

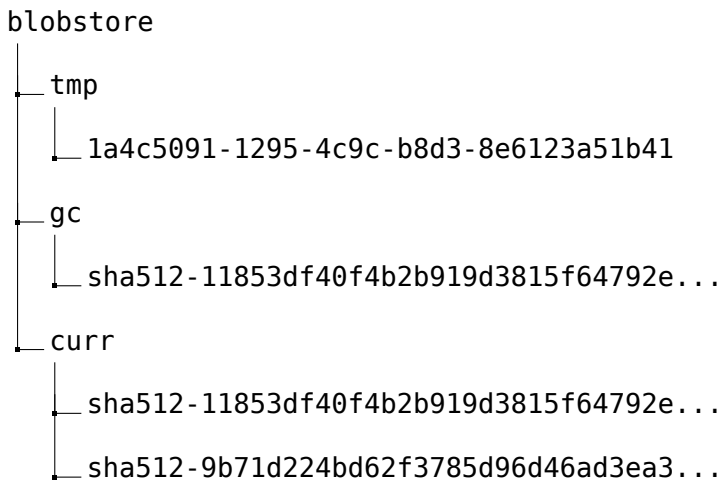
It is quite likely that the same blob would be shared by multiple "values" in the database. For a relational database these values are rows in a table, while for a document-oriented database, these values are documents. Hence, we provide an interface for garbage collecting the deleted blobs.

4.6.1 Starting the Garbage Collection

The `startGC` method takes a BlobStore as argument and starts garbage collection (GC) for that BlobStore. `startGC` does two things: It first creates the *gc* folder and then creates empty files in the *gc* folder corresponding to every blob in the *curr* folder. We use the `readdir` system call to iterate through the contents of the *curr* folder. Once a GC has started you can not start another GC on the same BlobStore

until the first one finishes - doing so will throw an error. Also, note that creation of new blobs and reading the old blobs can happen concurrently with the GC.

Figure 4.3: Directory structure of a BlobStore during GC



4.6.2 Marking a blob as accessible

Once a blob is marked as not deleted using the method `markAsAccessible`, we delete the file corresponding to that blob from `gc` directory. This ensures that the blob does not get deleted at the end of the GC.

4.6.3 End Garbage collection

This step involves removal of all the blobs which are not accessible. The `endGC` method takes a BlobStore as argument. It iterates over all files in the *gc* directory and deletes the corresponding blob from the *curr* directory. After this step finishes, `endGC` deletes the *gc* directory along with its contents.

4.7 Concurrency

In this section we will describe how the design described above allows concurrent access on blobs across processes.

Table 4.3: Interface for garbage collection

Methods	Purpose
`startGC`	Starts garbage collection for the given BlobStore
`markAsAccessible`	Marks the given blob as accessible
`endGC`	Ends the garbage collection by removing all the unaccessible blobs

The `newBlob` creates an empty file in the *tmp* folder. This method will never fail, as we ensure that the *tmp* folder is always present. In `endWrite`, we move the blob to *curr* folder. We use the `rename` system call for this, which is atomic i.e. a partial file will never be read [10]. Before moving the blob to *curr*, we mark it as accessible using `markAsAccessible`, otherwise it will get deleted when `endGC` is called.

`startGC` first creates an empty *gc* directory using `mkdir`. We rely on `mkdir` to be atomic i.e. if two processes try to create the *gc* directory simultaneously, only one of them will succeed. This ensures that at any instance, only one GC can be running. We then traverse the *curr* directory using `readdir` system call and create an empty file with same name in *gc* directory. Note that `readdir` is not thread-safe. However, in our implementation only one GC will be running at any instance, thread safety of `readdir` is immaterial.

In `endGC`, we traverse the *gc* directory and delete the corresponding file from *curr* directory. It might happen that a blob which was opened for reading by `startRead` gets deleted during `endGC`. We use the `unlink` system call for removing the file. `unlink` ensures that the file remains in existence until the last file descriptor referring to it is closed [15].

Chapter 5

Conclusion

5.1 Summary

In this thesis we described our design and implementation of Bloc - a library for handling large binary objects written in Haskell. We give a brief introduction to functional programming and describe various features of Haskell which we used in our implementation.

We also describe why programming with locks is difficult and how our design achieves concurrency without using any locks at the application level.

5.2 Future Work

Currently our code uses several functions which are supported only on POSIX compliant file systems. This means that our library will not work on Windows. In the future, we might look into adding support for Windows.

We are also working on creating a key-value store like Bitcask [16]. We plan to add support for storing `BlobIds` provided by our library as a value in the key-value store.

Proving that a system is concurrent and will work under all circumstances is very tough. In this thesis, we tried to give an "informal" explanation of various situations where we handle concurrency. In the future, we will try to give a formal proof of

concurrency of our design.

References

[1] Edgar F Codd. "A relational model of data for large shared data banks". In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[2] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: amazon's highly available key-value store". In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. ACM. 2007, pp. 205–220.

[3] Kristina Chodorow. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[4] Michael J Carey, David J DeWitt, Joel E Richardson, and Eugene J Shekita. *Object and file management in the EXODUS extensible database system*. University of Wisconsin-Madison. Computer Sciences Department, 1986.

[5] David Hows, Peter Membrey, and Eelco Plugge. "GridFS". In: *MongoDB Basics*. Springer, 2014, pp. 101–115.

[6] Russell Sears, Catharine Van Ingen, and Jim Gray. "To blob or not to blob: Large object storage in a database or a filesystem?" In: *arXiv preprint cs/0701168* (2007).

[7] Simon Peyton Jones. "Beautiful concurrency". In: *Beautiful Code: Leading Programmers Explain How They Think* (2007), pp. 385–406.

[8] Carl Hewitt, Peter Bishop, and Richard Steiger. "A universal modular actor formalism for artificial intelligence". In: *Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc. 1973, pp. 235–245.

[9] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. "Concurrent programming in ERLANG". In: (1993).

[10] *rename(2) man page*. URL: `http://man7.org/linux/man-pages/man2/rename.2.html` (visited on 05/30/2015).

[11] Daniel J Bernstein. *Using maildir format*. 1995.

[12] *mkdir(2) man page*. URL: `http://man7.org/linux/man-pages/man2/mkdir.2.html` (visited on 05/30/2015).

[13] John Hughes. "Why functional programming matters". In: *The computer journal* 32.2 (1989), pp. 98–107.

[14] Paul J Leach, Michael Mealling, and Rich Salz. "A universally unique identifier (uuid) urn namespace". In: (2005).

[15] *unlink(2) man page*. URL: http://man7.org/linux/man-pages/man2/unlink.2.html (visited on 05/30/2015).

[16] DS Justin Sheehy and D Smith. "Bitcask. a log-structured hash table for fast key/value data". In: *White paper, April* (2010).

[17] Mendel Rosenblum and John K Ousterhout. "The design and implementation of a log-structured file system". In: *ACM Transactions on Computer Systems (TOCS)* 10.1 (1992), pp. 26–52.