LibreS3: design, challenges, and steps toward reusable libraries

Edwin Török
Skylable Ltd.
edwin@skylable.com

LibreS3 is an Amazon S3 compatible server running on Ocsigen+Lwt, that uses a Skylable S^X cluster as storage backend. This presentation shows the architecture of LibreS3 and a set of libraries that result from splitting the application into reusable components. Finally it presents bugs and challenges encountered during its development. The core design principle is that you don't have to choose *one particular* monadic concurrency library, or *one particular* Xml/Json parser: you should be able to code your application as a functor for an abstract interface, and use either the predefined implementations, or provide your own where appropriate.

1 Concepts

LibreS3's architecture is not novel: it is based on existing, well-established concepts. What is interesting is how these all work together in a real application.

To understand the rest of this paper a basic understanding of the following concepts is useful:

1.1 Monads

Possible solution to write an application that handles multiple (potentially slow) events in parallel:

direct style block while each event is processed Use multiple processes for concurrency

threaded style as above but thread-safe and use system-level threading for concurrency [1]

asynchronous style with callbacks the usual solution in the imperative world (see `libev`)

monadic style the usual choice in functional programming, uses a monadic concurrency library

If the code is written in a monadic style it can still be instantiated to run in a blocking way (or with threads) for easier debugging, but changing code written in direct style to monadic style takes much more effort.

A monad tutorial is outside the scope of this presentation (the concept should be familiar to anyone who has used Lwt or Async), this section just summarizes the topic in an informally. Good starting points can be found in the bibliography [15] [12] [9] [6] [19] [18] [5] [1].

For the scope of this presentation we'll use a basic monad extended with exception handling (listing 1 on the following page). For a monadic concurrency library the α `t` value can be thought of as representing a deferred computation or a promise, and an application written in a monadic style is a chain of computations on them. The monad signature can be implemented directly by Lwt, and after a writing a few wrapper functions by Async too (although the semantics wrt to exception handling doesn't match exactly).

[1] only useful for I/O in OCaml, until the multicore runtime is ready

Submitted to:
OCaml meeting 2014

Listing 1: monad signature

```
(* the type of deferred computations *)
type +α t
(* immediate value to deferred value *)
val return : α → α t
(* chain computations *)
val ( >>= ) : α t → (α → β t) → β t
(* exception to deferred value *)
val fail : exn → α t
(* [try_bind m f g] either f or g *)
val try_bind : (unit → α t) → (α → β t) → (exn → β t) → β t
```

Monads don't have to represent deferred computations though, they can just be used for error handling, as in the result monad of listing 2. The implementation is straight forward, see full code samples in [21]. As we'll see in section 2.1 on page 4 a monad with exception handling can be transformed to the result monad for our purposes.

Listing 2: result signature

```
type ('ok, 'err) t
val return : 'ok → ('ok, 'err) t
val fail : 'err → ('ok, 'err) t
val (>>=) : (α, 'err) t → (α → (β, 'err) t) → (β, 'err) t
val catch : ('ok, 'err) t → ('err → ('ok, 'err) t) → ('ok, 'err) t
```

1.2 S3 server

Amazon provides a proprietary distributed object storage as part of its cloud offering, called an S3 server. It can be accessed from both inside and outside the Amazon cloud via an HTTP(S) REST API that is publicly documented [17].

Unfortunately the implementation is proprietary and is only provided as a service that runs inside Amazon's own infrastructure, which raises privacy and security concerns (aside from high running costs).

There are several FOSS implementations of the S3 protocol both client and server-side. LibreS3 is such a FOSS implementation for the server-side. That is you can run LibreS3 and have your existing S3 clients (`s3cmd`, `python-boto`, etc.) talk to it to store/retrieve/list objects.

The operations are rather simple on the API level, each request has an Authorization header that contains an HMAC signature for the current request:

retrieve an object from a bucket GET /bucketname/objectpath

upload an object to a bucket PUT /bucketname/objectpath

list objects in a bucket GET /bucketname ...

More complex operations are defined too (multipart uploads, ACL handling, etc.), the requests and replies use Xml (and sometimes Json) too.

The objects (that are files on the client) are stored in a distributed, redundant cluster. Redundancy is usually achieved by keeping multiple copies of the data (replicas).

1.3 S^X server

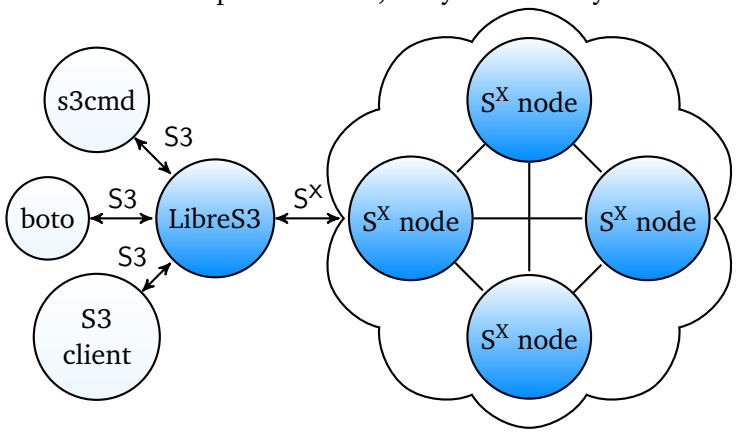
S^X [13] is a FOSS distributed cluster storage backend written in C designed for building private clouds that support deduplication, replication, garbage collection. S^X focuses on security, scalability, simplicity, speed, source-code access, and savings.

The API is HTTP(S) REST API, but unlike S3 it is designed around deduplicated storage: storing the same block of data only once. This has the advantage of reduced bandwidth usage, better resume handling, and that your data is immutable (only the metadata is mutable).

The disadvantage is that it is a far more complex system than a filesystem + rsync.

1.4 LibreS3

LibreS3 is an S3-compatible server, i.e. you can use your existing S3 clients to talk with it:



It implements the S3 protocol on the server-side, and acts as an S^X client that connects to a cluster of S^X nodes. The data is actually stored by S^X , LibreS3 just acts as a gateway/proxy. Most of the S3 APIs are implemented (those that can be mapped somewhat efficiently to S^X), while others are left out (like ACLs that behave differently in S3 and S^X).

Files uploaded by S3 clients are visible by SX clients and viceversa.

LibreS3 itself is stateless (when it does need to store state it stores it on the S^X cluster), so you can run multiple independent instances of LibreS3 without needing to synchronize them (that will happen on the S^X side).

2 Architecture and a useful set of libraries

The "core principle" described in the abstract originates from not wanting to choose one particular library/framework and be forever stuck with it. It is only respected for Lwt and Ocsigenserver interfaces in version 0.2 (not for parsing), and although internally LibreS3 is already separated into libraries, they are still tied to LibreS3 and not easily reusable elsewhere. There are some other problems like unit tests only exist for the protocol, debugging issues is sometimes hard as only printf-like debugging is possible, over-reliance on (often incomplete) stacktraces for tracking down problems.

For version 1.0 it would make sense to split these out and make them reusable [2]. This would make

[2] at the time of this writing this is a work in progress, library name are subject to change. Some of the libraries are already available as sample code.

testing easier and also make LibreS3 more useful/interesting to the OCaml community: a reusable set of libraries for writing HTTP(S) REST API servers, clients, parsing Xml/Json data, and an example application that uses it.

The general requirements would be: expose the minimum API as common type signatures for all implementations, functors to provide common higher-level features, implementations for direct-style/Lwt/Async/Cohttp/Mirage as subpackages (where possible), unit tests, minimum OCaml version 3.12.1.

This paper's title includes "steps toward reusable libraries", because some of these libraries are already available as sample code in [21], while others only exist at the design stage (i.e. they are too closely coupled with LibreS3 right now).

2.1 any-cache

This library illustrates the underlying concepts (monadic interface, functors, monad transformers) and processes (building, testing, packaging) quite well, although its implementation is quite simple (and could be used without the monadic part).

Certain operations (especially those involving the network) can be time consuming, so they would benefit from having their results cached. This library provides a fixed size in-memory cache, using a LRU/2Q [10] algorithm (similar to PostgreSQL's) which as its name says has two queues: one for frequently used values, and another for newly used values. If a value is not used again it will fall off the second queue, otherwise it is promoted to the main queue. This ensures that massive sequential fetches don't evict otherwise frequently used values from the cache. Values are not evicted from the cache directly, instead they are placed in a victim cache that will remember the keys. If a value is requested from the victim cache it is placed directly into the main queue: this allows a much larger window to detect requests for same value without actually storing those values in memory.

Since OCaml is a garbage-collected language the values won't actually be deleted from memory when they are removed from the main cache and it would be a waste to recompute a value that we have already. Hence the victim cache is actually a fixed size **Weak** map of keys to values. If a value hasn't been removed by the GC yet then we can return it directly, otherwise we return just the key as the original algorithm would.

Compared to a purely **Weak** map based solution this has the benefit of having a predictable ensured cache size (we always store the values on the main and secondary queues in memory), and still benefits from returning values that haven't been deleted by the GC (with just a **Weak** map you could loose all values as soon as a GC cycle is run).

Listing 3: LRUCacheResult.mli

```
module type Result = sig
  type ('ok, 'err) t
  val return : 'ok → ('ok, 'err) t
  val fail : 'err → ('ok, 'err) t
  val (>>=) : (α, 'err) t → (α → (β, 'err) t) → (β, 'err) t
  val catch : ('ok, 'err) t → ('err → ('ok, 'err) t) → ('ok, 'err) t
end
module Make(R:Result) : sig
type ('ok, 'err) t
val create : int → ('ok, 'err) t
```

```
val get : ('ok, 'err) t → notfound:'err → string → ('ok, 'err) R.t
val set : ('ok, 'err) t → string → ('ok, 'err) R.t → unit
val lookup : ('ok, 'err) t → string →
        (string → ('ok, 'err) R.t) → ('ok, 'err) R.t
end
```

There is a functorial interface described by listing 3 on the preceding page that expresses caching as computations on the **Result** monad. There is an interface for direct access to the cache: `create` with specified size, `set` a new value in the cache and remove least-recently used value(s) to make room and `get` a value from the cache (updating the LRU statistics as a side-effect). These can cache both successful results and errors. Note that keys are always strings, this limitation could be removed with an additional functor parameter similar to **Map.Make**, but that would only work for in-memory caches. If we want to use a disk or network-based cache in the future the key would have to be serialized anyway.

A higher-level `lookup` function is provided that tries to retrieve a value from the cache, and if it doesn't succeed (either because the value is not in the cache, or the previous computation resulted in an error) then the value is (re)computed and the result stored in the cache. If the computation result is an error it will be cached, but the next `lookup` will attempt to compute it again. This is useful if the computation involves a network call.

Although this functorial interface is well suited for the implementation a simpler interface is provided too in listing 4 as an instantiation of the above functor:

Listing 4: LRUCache.mli

```
( ... )
val lookup :  ('ok, 'err) t →
        string → (string → ('ok, 'err) result) → ('ok, 'err) result
```

LibreS3 code is written in a monadic style to express deferred computations (mostly network I/O), so a more useful functor is one based around an exception monad that extends the basic monad with a way to express failure, and a way to catch errors:

Listing 5: LRUCacheMonad.mli

```
module type TryMonad = sig
  include Monad
  val fail : exn → α t
  val try_bind : (unit → α t) → (α → β t) → (exn → β t) → β t
end
module Make(M:TryMonad) : sig
( ... )
val lookup_exn: (α, exn) t →
        string → (string → α M.t) → α M.t
end
```

This is internally transformed to a **Result** monad by an internal monad transformer and then we reuse the functor from listing 3 on the preceding page to implement `lookup` and `lookup_exn`. Lifting a monadic computation to the **Result** monad is fairly straight-forward:

```
let lift f v = M.try_bind (fun () → f v) return fail
```

The monadic interface raises a new question: what happens when we want to compute a value that is not in the cache, but for which we've already launched a computation. The answer is that the cache stores pending computations: when the pending computation finishes (becomes determined) the value is returned by the monadic `get` function. If we stored a failure we attempt the computation again, otherwise we return the successful result.

This has the advantage that a slow / resource intensive computation doesn't drain additional resources if it is requested multiple times before it completes. A downside is that the `lookup` function is not completely safe: you can trick it into returning a value that never becomes determined [3] by doing a nested lookup for the same cache key. The equivalent in direct mode (non-monadic) code would be an infinite (mutual-)recursion, except that you would get an exception in that case (and likely your program killed).

2.1.1 Testing

The test is written as a functor using `OUnit`, and instantiated using 3 implementations for the **TryMonad**: two based on the monadic concurrency libraries **Lwt** and **Async**, and a third based on an immediate evaluation monad. Code is compiled both in bytecode and native code mode (if available).

Testing with different concurrency libraries is useful because their semantics is slightly different (especially concerning how far they execute values immediately and when they defer the computations), and testing with the immediate mode is useful because it provides a determistic testcase.

The downside is that this involves some boilerplate that has to be repeated for each library that you want tested this way (in terms of build instructions, and instantiations of the functors).

2.1.2 Initial code from LibreS3

Initially the **Cache** module provided a different function trying to simulate a monadic bind function, however that had a few shortcomings: an extra parameter specifying which cache use, it wasn't polymorphic in the input parameter and in practice the input was always an **E**.t, not a monad:

```
val bind : E.t M.t → (E.t → α M.t) → Cache.t → α M.t
```

2.1.3 Packaging

The build system uses ocamlbuild, and it is generated by OASIS [4]. The monadic style takes its toll on having to write boilerplate code in each library (compare `any-http` and `any-cache`, they are quite similar) to build the unit tests with each supported monad implementation both as native and byte-code.

2.2 any-io

In version 0.2 there is the usual type-signature for a monad with exception handling, a monad for detached computation, an interface to a "Unix" monad. All these have 2 implementations: one using Lwt/Lwt_unix, and one using direct-style with the Unix module.

[3] of course this won't actually hang the whole program when using a monadic concurrency library

[4] some may consider it an overkill, but I believe that standardizing and improving on one build system is better than having scattered/incomplete custom build systems

There is also an attempt at a somewhat higher-level file-based IO, but LibreS3 and S^X are an object storage and the APIs don't fit well. The new library should provide functors and type signatures for: the usual monad, with exception handling; an extended monad for detached computations; low-level monad wrapping file I/O from Unix module; a higher-level monad with a key-value store interface, temporary storage handling, etc.; and a functor to build the higher-level monad from the low-level ones.

2.3 any-http

A library that allows you to use OCamlnet, Ocsigen_http_com/Ocsigenserver, or Cohttp as the underlying implementation. The concept is not novel: Mirage provides something similar. In fact the Cohttp API is almost good for this purpose, however there are some inconsistencies between the Async and Lwt APIs so they can't match the same type signature. This library is modeled after the Cohttp API, and provides all 3 implementations underneath. In the future it could also provide a builtin (but customizable) way to handle authentication, errors/exceptions, and limits.

Listing 6: example for use of any-http library

```
module MyApp(H : Httpintf.S) = struct (* ... *)
let uri = Uri.make ~scheme:"http" ~host ~port ~path () in
H.Client.call meth uri (H.Headers.init ()) (H.Body.empty)
>>= fun (status, headers, bodystream) →
H.Body.to_string bodystream >>= fun body →
(* ... *)
module App = MyApp(Httpservice_cohttp_async)
module App = MyApp(Httpservice_ocsigenserver)
module App = MyApp(Httpservice_cohttp_lwt.Make
  (Cohttp_lwt_unix.Server) (Cohttp_lwt_unix.Client))
```

You write your application (`MyApp` in this case) as a functor that takes the HTTP implementation (not the underlying monad) as a parameter. At the end you can instantiate your application with the actual implementation you want.

Provided are:

Httpservice_cohttp_async ocamlfind package `httpservice.cohttp.async`

Httpservice_ocsigenserver ocamlfind package `httpservice.lwt.ocsigenserver`

Provides a module as an Ocsigenserver extension, and also provides a client implementation.

Httpservice_cohttp_lwt ocamlfind package `httpservice.lwt.cohttp`

Can be instantiated with the Unix implementation of Cohttp, but might be useful for Mirage as well

Not provided in the library, but used in LibreS3, is an implementation based on OCamlNet: this uses a separate thread and a multiplexed pipeline, and binds to the main thread by a Lwt value (using the thread-safe Lwt primitives to wake up the main thread). A better implementation would be to use a monad based on Equeue, and perhaps nethttpd for the server side.

This library could be useful as a simple example on how to write a somewhat more complicated monadic application, and how to organize the user-facing interface: the functor taking a generic monad, and instantiations for commonly used monads (Lwt, Async, etc.).

It could be a useful exercise to use this library for cross-testing Lwt/Async code in Cohttp, or Ocsigenserver and Cohttp (i.e. can Cohttp&Async talk to Lwt&Ocsigenserver in all situations?).

2.4 ioconvcomb

Motivation: often there is some library / code generator that deals with a file format, but that uses a different Json library than your preferred one. If you build a large application you can easily end up with several Json implementations due to these different requirements (for example Yojson vs Jsonm).

Writing parser / printers by hand is tedious, lacks consistent error handling on syntax / type mismatches. Some articles show a way to unify parsing and printing such as [4] [11] [16], however I think the most important part is separating the type / shape description from the actual parsing. That way you can use the same type description with different parser implementations. None of this exists in version 0.2, all the parsing is hand-written for Xmlm and Jsonm.

This library should provide a way to define type isomorphisms (either manually, or via a generator like atdgen/piqi). And then provide a Json, Xml, Sexp, etc. parser/printer and define an isomorphism between Json and your type for example. The choice of the Json/Xml parser won't be fixed, you could provide your own if you don't like the default. It would also provide consistent error handling (file position for syntax errors, tree path for semantic errors, etc.), and be useful in writing unit tests as any server is automatically a client too, at least at the protocol level!

This library would be similar to [3], [2] and [14], in fact it should probably leverage those libraries, instead of reinventing the wheel, and provide only the pluggable Json/Xml implementation.

2.5 S^X client

This is a pure-OCaml implementation, it doesn't use the C libsx library (which doesn't support custom event loops). In version 0.2 this is provided as an implementation of the file-like API, however for 1.0 it should be a separate library providing direct access the entire S^X REST API.

3 Development challenges

I encountered a few bugs in some libraries for which I provided patches, and collaboration with upstream was successful: LibreS3 works now correctly with unpatched upstream libraries!

One major bug[5] was a double-release of the OCaml runtime lock in OCamlnet's SSL code, which was already reported on the ML. I tracked it down using a debug patch for OCaml's thread runtime [7]. It is yet to be determined what the fate of this patch should be: should it stay as a patch (perhaps used via an opam switch) whenever you want to debug such crashes, or should it be integrated in OCaml trunk?

Another bug manifested itself on the day of the first release: the application hung due to an incorrect use of Unix.fork (instead of Lwt_unix.fork) in Ocsigenserver. Trying to work it around I introduced a bug in LibreS3 itself [20], which was properly fixed in version 0.2.

Some other bugs: OCamlnet too strict URL handling, Lwt readdir crash, OCamlnet SSL persistent connections lost, ocamlbuild SIGPIPE ocaml-ssl runtime-lock double-release.

[5] well, less serious than the Dom0 crash found when trying out Mirage [8]

While ocsigenserver is great when used for what it was designed for (a webserver, optionally hosting applications), it is not simple to use as an embedded webserver as there is no way to fully configure it from the API. I wrote a patch to add HTTP PUT and DELETE support — both essential in implement a REST-like server. Also the default limits are somewhat high: 1000 worker threads for Lwt which never exit (decreased to 64 in LibreS3).

Writing code in monadic style has a disadvantage too: debugging is notoriously hard. Using the blocking implementations, and splitting the code into individually testable modules (as outlined by this paper) helps somewhat. However since most of the functions are tail-recursive now the stacktraces will lack important information needed to fully understand an exception. There is no good solution for this for a functional program, short of better logging, or some way of naming the scope of function calls that could be passed down to a logger.

All the major bugs were due to code that interfaces with C or the operating system, and not with the OCaml code itself, which made pinpointing them and debugging somewhat easier as the usual C tools could be used for that (valgrind, strace, tcpdump, etc.).

That doesn't mean that the OCaml code was (or is) without bugs, the myth that "if it compiles it will run correctly" should be avoided. Certainly the type system can be used to reduce the amount and complexity of bugs encountered during runtime, but for that the concepts needs to be encoded in the type system correctly. For example although percent-encoded URLs and percent-decoded URLs are both strings you must only run that operation only once on a URL, otherwise you introduce a bug, hence using separate types would be encouraged. Care must be taken when dealing with resources, it is recommended to encapsulate operations that require acquiring and releasing resources (such as opening and closing a file) in a (monadic) function, `with_resource` to ensure they are always closed even if exceptions are encountered, and that the error from the close doesn't hide the initial exception. Of course none of these mechanisms are perfect, and they can be escaped, but they help reducing the number of bugs in general.

3.1 Build and packaging woes

The situation is easy enough if the user has (or can install) OPAM, however that is not always possible ...

It would be good if there was a tool that would generate package build rules for the major OS distributions (Debian, RHEL/CentOS/Fedora, FreeBSD). That is what our target audience uses, for example sysadmins in enterprises usually want CentOS 6 binary packages.

The main problem is the lack of a "new enough" version of the OCaml compiler, and backporting is not sustainable: 3.11.2 is mostly unsupported by libraries by now, and 3.12.1 starts to get problematic (`ocaml-cstruct` and `ocaml-dns`).

In some situations it might be possible to build "portable Linux binaries", (see the `4.01.0+lsb`, `4.01.0+musl+static` opam switches), but they were not suitable for LibreS3 [6]. However in general it is better to build distribution specific packages where possible.

Providing a source tarball that builds everywhere without requiring the user to manually compile and install a lot of external dependencies is a challenge. The approach in 0.1 (custom build script + embed dependent libraries) is very slow, and requires manual intervention whenever dependencies change with a new library release.

[6] they may be suitable for opam itself though to solve the chicken and egg issue

4 Additional information

This paper, the presentation poster, sample source code and slides are available at [21]. Thanks to the reviewers of the extended abstract for their feedback.

References

[1] Chris Barker & Jim Pryor (2010): *NYU Lambda Seminar*. Available at http://lambda.jimpryor.net/week7/.

[2] camlspotter: *type_conv for various tree data formats.* Available at https://bitbucket.org/camlspotter/meta_conv.

[3] Simon Cruanes: *combinators for type conversion (serialization/deserialization, with GADTs)*. Available at https://github.com/c-cube/cconv.

[4] Simon Cruanes (2014): *Universal Serialization and Deserialization*. Available at http://cedeela.fr/universal-serialization-and-deserialization.html.

[5] Jérémie Dimino (2012): *Lwt user manual*. Available at http://ocsigen.org/download/lwt-manual.pdf.

[6] Jake Donham (2009): *Equeue compared to Lwt*. Available at http://ambassadortothecomputers.blogspot.ro/2009/02/equeue-compared-to-lwt.html.

[7] Török Edwin (2013): *OCaml: debug mode for native otherlibs/systhreads*. Available at http://caml.inria.fr/mantis/view.php?id=6204.

[8] Török Edwin (2014): *Linux netback crash trying to disable due to malformed packet*. Available at http://xenbits.xen.org/xsa/advisory-90.html.

[9] Brian Hurt (2007): *A Monad Tutorial for Ocaml*. Available at http://blog.enfranchisedmind.com/2007/08/a-monad-tutorial-for-ocaml/.

[10] Theodore Johnson & Dennis Shasha (1994): *2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm*. In: *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 439–450. Available at http://www.inf.fu-berlin.de/lehre/WS10/DBS-Tech/Reader/2QBufferManagement.pdf.

[11] Oleg Kiselyov (2009): *Type-safe functional formatted IO*. Available at http://okmij.org/ftp/typed-formatting/.

[12] Xavier Leroy (2014): *Monadic transformations, monadic programming*. Available at http://pauillac.inria.fr/~xleroy/mpri/2-4/monads.2up.pdf.

[13] Skylable Ltd. (2014): *Skylable S^X*. Available at http://www.skylable.com/products/sx.

[14] Anil Madhavapeddy & Thomas Gazagnaire: *Dynamic types for OCaml*. Available at https://github.com/mirage/dyntype.

[15] Yaron Minsky (2013): *Real world OCaml*. O'Reilly Media, Sebastopol, CA. Available at http://www.realworldocaml.org.

[16] Tillmann Rendel & Klaus Ostermann (2010): *Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing*. In: *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, ACM, New York, NY, USA, pp. 1–12, doi:10.1145/1863523.1863525. Available at http://www.informatik.uni-marburg.de/~rendel/unparse/rendel10invertible.pdf.

[17] (2006): *Amazon S3 REST API documentation*. Available at http://docs.aws.amazon.com/AmazonS3/latest/API/APIRest.html.

[18] Mark Simpson (2012): *From Functor to Applicative*. Available at http://blog.0branch.com/posts/2012-03-26-02-from-functor.html.

[19] Mark Simpson (2012): *Implementing Functor in OCaml*. Available at http://blog.0branch.com/posts/2012-03-26-01-implementing-functor-ocaml.html.

[20] Tobias Taschner (2014): *File upload error after a few uploads*. Available at https://bugzilla.skylable.com/show_bug.cgi?id=482.

[21] Edwin Török (2014): *Additional materials (sample source code, poster, slides)*. Available at http://goo.gl/jmFOcn.