Dimitar Popov

Concurrent revisions library for OCaml

Part II of the Computer Science Tripos

Homerton College

February 26, 2014

Proforma

Name:	**Dimitar Popov**
College:	**Homerton College**
Project Title:	**Concurrent revisions library for OCaml**
Examination:	**Part II of the Computer Science Tripos, July 2014**
Word Count:	
Project Originator:	Dr Anil Madhavapeddy
Supervisor:	Dr Anil Madhavapeddy

Original Aims of the Project

To design and build a library for OCaml that implements the concept of Concurrent revisions. Test the library and implement use cases using the library. Understand the trade offs both between the different paths that can be chosen during the implementation of the library and between the more traditional means of concurrent programming and the concept at hand. Evaluate the differences between the API of the original implementation written in C# and the more functional one that is natural to OCaml.

Work Completed

Special Difficulties

Declaration

I, Dimitar Popov of Homerton College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Acknowledgements

Chapter 1

Introduction

1.1 Overview of the project

The biggest challenge when using parallel programming is typically how to keep track of the side effects of computations that are executed in parallel. Traditional method for dealing with this issue often limit concurrency, do not provide sufficient determinism and are error prone. Ideally, we would like a concept where all conflict between parallel tasks are resolved deterministically with as less as possible effort from the programmer.

One concept that satisfies these requirements is that of Concurrent Revisions, initially proposed at OOPSLA'10 [1]. The aim of this projec is to implement this concept in the functional language OCaml and evaluate its performance and usability. The domain of functional languages was chosen because of their inherited determinism which makes using parallelism less complex and provides a facility for tracking side effects. I have designed and implemented a library that incorporates the ideas of Concurrent Revisions and ensured its correctness with a number of unit tests. Together with some small example code, two use cases were produced using the library - a logging system and a chat service. They were used to evaluate the performance and usability of the implementation and the whole concept in the world of OCaml. The conclusion was that [add when actually have a conclusion].

1.2 Overview of Concurrent revisions

The idea of Concurrent revisions as initially proposed highlights three main design choices:

- **Declarative data sharing** - the user declares what data is to be shared between parallel tasks by the use of isolation types.
- **Automatic isolation** - each task has its own private stable copy of the data that is created at the time of the fork.
- **Deterministic conflict resolution** - the user also specifies a merge function that is used to resolve write-write conflicts at that might arise when joining parallel tasks. Given that this function is deterministic, the conflict resolution is also deterministic.

In this framework the unit of concurrency are asynchronous tasks called revisions. They provide the typical functionality for asynchronous tasks - the user can create, fork and join them. This removes the complexity of synchronization out of the tasks themselves and gathers in into a single place - the merge function.

1.3 Motivation

1.3.1 Overview of other approaches to concurrency

Concurrency is essential and vital in multi core architectures and in distributed systems. Traditional approaches rely on synchronizing parallel tasks by locks, event driven formalisms or similar. This makes conflicts very expensive if determinism is needed. Moreover, these methods are often error prone and extremely hard to debug.

Standard locking schemes are sometimes a good approach to ensure consistency of data shared between multiple parallel tasks. However locking limits concurrency since task are blocked until it is safe to continue. Significant effort is required from the programmer to reason about all possible interleaves of task execution. Identifying the scope of critical sections becomes tricky as it could either limit concurrency or provide insufficient isolation. This approach is highly error prone and extremely difficult to maintain.

Instead of locking one can use event-driven systems, where tasks execution is triggered by event from other tasks. This results in inverted control structure of the program. The programmer's control flow becomes inverted as well and results in convoluted control logic. In such a system, often the actual tasks have to be very fine grained in order to maximize performance which complicates the logic and makes it difficult to maintain.

Another approach is instead of trying to avoid conflicts to try to resolve them. This is in the core of transactional systems in which each tasks takes a copy of the shared data and conflicts are resolved at the time of the join. However conflicts are resolved non-deterministically which complicates reasoning about the execution. Another criticism of transactional systems is that they ensure serializability, which is not necessary for all use cases and limits concurrency[4]. Moreover they ofter rely on roll-backs in order to deal with conflicts which means that a lot of work is throws away and repeated afterwards in the hope that this time there will not a conflict which is wasteful.

In the concept of concurrent revisions the guarantee of parallel executions being equivalent to some sequential schedule is relaxed. Instead, given the right abstractions, the programmer can reason about the execution directly.

1.3.2 The contribution of Concurrent revisions

Much like transactional systems, Concurrent revisions use replication to ensure isolation. Because of that roll-back of aborted revisions comes for free, they are also relatively rare events since in most cases the conflict can be resolved. The guarantee of parallel executions being equivalent to some sequential schedule is relaxed. This increases parallelism and leaves to the programmer only two things to worry about - what has to be shared and how conflicts have to be resolved, concentrating any possible bugs in a limited region. With Concurrent revisions, given the right abstractions, the programmer can reason about the execution directly making the design of a concurrent system more natural.

This approach is data centric in a sense that it takes the complexity of synchronization, in terms of programmer's effort, out of the tasks and adds it to the data declarations. The runtime complexity of synchronization is shifted from blocking or checking that the schedule of tasks was legal into the join of tasks where conflicts are resolved by a deterministic computation.

1.3.3 Applicable areas

Every system that is subject to a lot of conflicts typically has to limit the parallelism of its execution in order to ensure consistency and avoid conflicts. Concurrent revisions take the different approach of resolving conflicts instead of avoiding them by scheduling. This makes them suitable for problems where there are a lot of write-write conflicts which should be resolved deterministically and performance can be increased greatly by more parallelism. Some examples of applications that could benefit from concurrent revisions are:

- **Bank transactional systems** - Such systems have a lot of constraints on invariants that form write- and read-skews. We will see later how this can nicely be resolved if using concurrent revisions (see 1.5.3).
- **Games** - They are a natural example when high parallelism is crucial for adequate performance. However, the fact that there are a lot of conflicts-user input, graphics rendering, simulating physics, game logic, write-backs to disk, makes getting their parallelization right tricky. Now what if we execute each of these tasks in separate revision and then join them as appropriate. There is one more concern of course - we have to be able to resolve conflicts. Luckily in order to do so, we simple have to define the merge function, which bundles all the complexity of dealing with conflicts into a single place, making it much more maintainable. Getting the merge function right is crucial, as we do not want our player to dash in a wall that was not displayed on the screen yet.
- **Logging & Chat systems** - The usage of functional languages for large commercial systems is increasing in large distributed systems such as logging and chat systems. One example of that is the Facebook chat, which is written in Erlang. Such systems are often large and distributed and have a lot of conflicts, timing and consistency are vital and they more or less require deterministic behaviour. This matches the list of requirements for suitability of concurrent revisions and we will see later that it is indeed convenient to write such systems using them.

1.3.4 Why OCaml?

OCaml is functional programming language that is getting increasingly more popular both in academia and in the industry. The increasing amount of libraries for OCaml makes it an excellent choice for a variety of use cases.

As a functional language it provides a natural means of tracking executions using type checking and immutable data structures. Replication of complex immutable data structures in OCaml are very cheap, since no actual replication is done, but rather upon updates the structure of the old value is heavily reused, which makes updates cost only constant space.

These features of OCaml makes it a very efficient environment for implementation of Concurrent Revisions.

One down-side of OCaml is its limited parallelism. The run-time system is single threaded which means that there is only one parallel task ever in execution. There is no guarantee on the scheduling and interleaving of tasks which makes it non-trivial to writer parallel software in OCaml, similar to the majority of programming languages. Due to this fact, there is no performance improvement due to exploring hardware parallelism expected when using revisions. Nonetheless blocking and/or wasting CPU time can be reduced significantly by them. The key benefits of using revisions are better responsiveness at the cost of little overhead and decreased amount of effort required by the programmer.

1.4 Quick overview of OCaml, the Core and Async libraries

OCaml is a garbage-collected functional language that also has object-oriented features. This section gives a brief overview of the features of OCaml used in the project.

1.4.1 Basic Types

As every functional language OCaml has a strong-type system that is incredibly useful in spotting bugs at compile type. The most heavily used types in OCaml are immutable. Here is an example of these:

```
1  let x = 1
2
3  let x = 2 in
4    print_int(x)
5
6  let z = ("Hello", 1, 3.14)
7
8  let l = [1,2,3]
```

On line 1 the user declares the variable x and assigns it the value of 1. This value is immutable and cannot be assigned to again. It can only be shadowed by another variable of the same name. The type checker resolves the type of x as int. The let binding specifies the scope of the declaration. In this case the variable x has a scope form line 1 to the end of the program. The let ... in binding allows us to declare a scope for the variable. On line 3 you can see that x is shadowed by another variable with scope until the end of line 4. There are also tuple data types, an example of which you can see on line 6. Here z contains 3 values of different types. The type of z is string*int*float. On line 8 we can see an example of a list. Lists in OCaml are implemented as a single-linked lists and a pointer to the head of the lists. This makes most access and update operations on list linear in time. Lists are also immutable. Updates reuse the structure of the old list and only replace the updated values, making them very space efficient.

Another important set of types are the functional types. Here is an example:

```
let add a b = a + b

let rec factorial n = n * (factorial (n-1))
```

The function add is of type int -> int -> int, takes two integer arguments and returns one integer result. The function factorial is a recursive function. Recursion is very important in functional languages as the data structures drive the design of the system build in a functional manner and they are typically hierarchical, making it natural to operate on them in a recursive manner. We can have functions from every OCaml type to every OCaml type as well as polymorphic functions.

Other basic structures include variants and records:

```
type point = { x : float; y : float }

type option = None
             |Some of `a
```

Each variable of type point has two fields y and x both of type integer. This is called a record type. The type option is a variant type. A variable of type option can either be equal to None or Some(x). Notice the type ‘a - it specifies a polymorphic type which allows x to be of any type.

OCaml also has imperative features:

```
1 let x = ref 1
2
3 x := !x + 2
```

Here x is declared as a int ref and is a reference to a particular place in memory. The value of x itself cannot be changed. What can be changes of the value of the place where it points to. On line 3 x is updated by assigning to it the sum of the dereferenced value of x and 2.

1.4.2 Complex data structures from the Core library

The Core library is a wrapper around the standard OCaml library that provides additional features. It this project I used the map and set data structures which are implemented as AVL trees. An AVL tree is a self-balanced binary search tree that guarantees a logarithmic complexity for insertions, deletions and updates due to its balanced structure. Both these data structures are immutable, allowing them to share great proportion of their internal structure. This makes replication cheap both in terms of space and time.

1.4.3 Module system

The module system is a key feature of OCaml. It can be used to package together related definitions. For example one can package a particular data type together with the associated operations over that type and abstract away the actual implementation of that type. Here is an example of a simple module:

```
1 module Balance : sig
2   type t
3   val add: t -> t -> t
4   val of_int: int -> t
5 end = struct
6   type t = int
7   let add a b = a + b
8   let of_int x = x
9 end
```

Here from line 1 to 4 is declared the signature to the module `Balance`. This is the interface available when using the module. The actual implementation of the module is from line 5 to 9. It has to match the signature of the module.

A valuable feature when dealing with modules are the functors. Functors can be seen as functions from modules to modules. Here is an example of functor usage:

```
1 module IntSet =
2    Set.Make(struct
3              type t = int
4              let compare x y = Int.compare x y
5             end)
```

Here I am using the build-in in Core functor `Set.Make` to create an integer set. The functor expect a module with a signature that requires the type of the set elements, i.e `t`, and a comparison function between elements. Notice the use of the build-in comparison function from the module `Int` on line 4.

1.4.4 The Async concurrency library

The Async library was used as the exclusive source of parallelism throughout the project. Async is a monadic concurrency library. It is build around the idea of deferred computations that are scheduled non-deterministically. It has a global lock that ensures only one computation will be in execution at any given time. Each computation is executed atomically which guarantees two computations will never overlap. The most common pattern for programming with Async is to schedule new computations to be executed in an event-driven fashion over the outputs of a previous computation once they are determined. This results in a guarantee for a sequence of actions to be performed in a particular order. Example taken from [3]:

```
1 Reader.file_contents filename
2    >>| fun text ->
3    List.length (String.split text ~on:’\n’)
```

Here on line 1 we schedule a deferred computation that reads the context of a file. On line 2 we bind it to a function that will be scheduled after the output of the read operation is determined, upon which it computes the number of lines in the file.

This pattern can be used to overcome in event-driven manner the problem of the blocking nature of reading and writing to streams, when they are respectively empty or full.

1.4.5 Additional reference

For more detailed overview of OCaml please see **Real World OCaml - Jason Hickey, Anil Madhavapeddy, and Yaron Minsky; O’Reilly 2013** [3], which provides an excellent and very approachable introduction to OCaml.

1.5 Deeper look into the concept

1.5.1 Data structures & Runtime behaviour

The main data structure in the concept are the revisions. They can be seen as a stable context for each asynchronous task as they are isolated of each other. The isolation types encapsulate the structure of the data to be shared.

Let’s look at a pseudo-code simple example:

```
IntIsolated = isolate(int)
IntRevision = Revision.make(IntIsolated,
3                     fun head parent current -> head + current - parent)
4  (account, revision) = IntRevision.create 0
5
6  let rev1 = revision.fork(fun r -> account = account + 5)
7  let rev2 = revision.fork(fun r -> account = account + 10)
8
9  assert(account in revision = 0)
10 assert(account in rev1 = 5)
11 assert(account in rev2 = 10)
12
13 let rev_join1 = join rev rev1
14 let rev join2 = join rev_join1 rev2
15
16 assert(account in rev_join1 = 5)
17 assert(account in rev_join2 = 15)
```

Example 1.

On line 1 the programmer creates a isolation type that isolates the primitive type integer. Then on line 2 and 3, he creates a `IntRevision` module by specifying the isolated type and the merge function. This function takes three arguments - the value of the isolated in the revision we are joining to, the value at the time of the fork and the current value in the joinee. Then he creates a revision specifying initial value for the isolated to be 0. This returns a tuple with type `IntIsolated.t * IntRevision.t`. The user than can use `account` to access its value in different revisions.

One line 6 and 7 we two new revisions are forked. Each would credit the account with 5 and 10 pounds respectively. At this point `account` has different values in each of the three revisions.

Then the two new revisions, returned by the forks, are joined one by one to the main initial revision (line 13 & 14). Luckily due to how the merge function is specified and the deterministic nature of the concepts, the account has the right amount at the end - 15 pounds.

If we have used a more traditional approach, we would have had to lock the whole system each time we access the value of the account or roll-back and redo the second fork. With revisions we simply synchronize the tasks when we join them.

Revision diagrams are an intuitive way for representing graphically the revision flow. In Fig.1 is an example of a revision diagram for Example 1.

1.5.2 Illegal revision diagrams

Not all possible joins are legal as some of them might invalidate the assumptions about the flow of revisions.

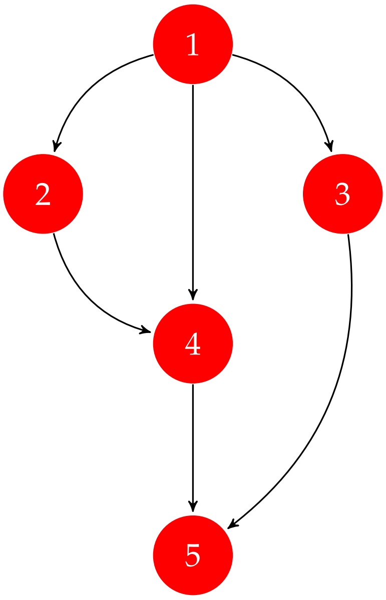

Fig.1: Revision diagram of Example 1. In the following diagram the nodes are the revisions. Outgoing arrows represent forks and joins are represented by incoming arrows, which are always two, one straight for the revision we are joining to and one bend for the joinee. In the diagram nodes correspond to revisions as follows: 1 - `revision` 2 - `rev1` 3 - `rev2` 4 - `join_rev1` 5 - `join_rev2`

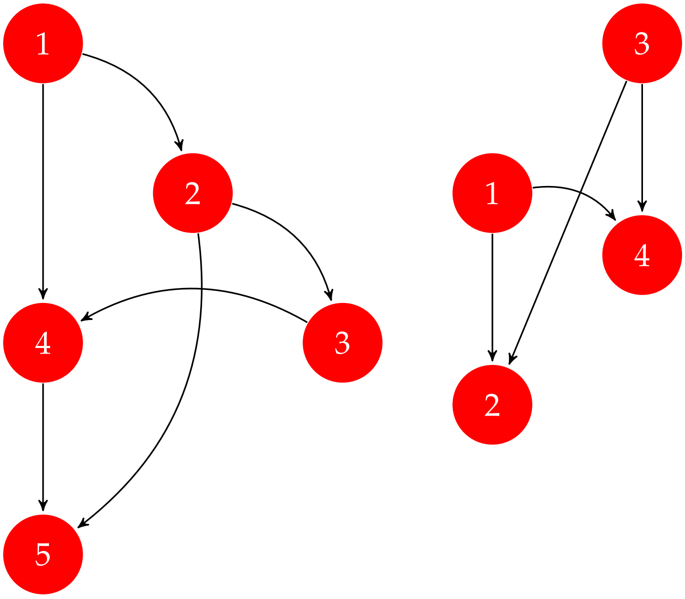

Fig.2: Illegal revision diagrams.

In Fig.2 we can see two illegal revision diagrams. In the first one, we join revision 3 to revision 1 before we have joined revision 2 which is the parent revision of 3. This means that the result in revision 5 might not be what we would expect since it is unknown how much of the work done in the fork for revision was done before forking revision 3. When we join 2, some of the changes occurring (which might have subsequently been modified) in 2 are already joined when 3 was joined. Since the merge function cannot account for such an interleaving as it is chosen by the programmer arbitrarily, such a revision diagram is not valid. One can imagine a more complex concept where this is accounted for at the time of the join. However this would require significantly more programming effort when designing the merge function. This would result in more difficult to reason about concept, prone to programming error, without adding extra functionality, since the legal revision diagrams are already expressive enough.

In the second example, we are interleaving two separate flows of revisions and there is no way to ensure that they isolate a similar state. This is completely meaningless, since they could isolate different variables and represent unrelated states.

1.5.3 Concurrent revisions in the world of OCaml

The OCaml implementation of Concurrent Revisions is presented in chapter 3. One of its advantages over the C# implementation is that it is purely functional and the type checker catches some of the typical mistakes that can be made - trying to access an isolated of a wrong type or join revisions of different types. What it does not do however is check for all types of illegal revision diagrams, instead an exception is raised whenever an illegal join is executed. This is far from ideal and statically type-checking joins for compatibility could be implemented as a future extension.

The API that the implementation exposes to the user is intuitive and resembles the typical OCaml approach for APIs for external libraries. Here is a simple example of its usage:

```ocaml
module IntRevision = Make(struct
    type t = int
    let merge head parent current = head + current - parent
  end)

let () =
  let r = IntRevision.init () in
  let res1 = IntRevision.create r 0 in
  let revision = IntRevision.get_revision res1
   and account = IntRevision.get_isolated res1 in
     Deferred.both
      (IntRevision.fork revision
        (fun r -> return (IntRevision.write r account ((IntRevision.read r account) + 5)))
      (IntRevision.fork revision
15          (fun r -> return (IntRevision.write r account ((IntRevision.read r account) + 10)))
16       >>|(fun (rev1, rev2 ->
17          let join_rev1 = IntRevision.join revision rev1 in
18          let join_rev2 = Intrevision.join join_rev1 rev2 in
19            assert(IntRevision.read join_rev2 = 15)
```

Example 2.

In Example 2 we can see the actual implementation of the pseudo code in Example 1. From line 1 to 4 we create the `IntRevision` module using a simple anonymous module specifying the type of the isolated data and the merge function. Then on line 7 we initialize an empty revision and on 8 we add a new isolated to the initial revision to create a new one. We then fork the two asynchronous account credits (line 11 to 14) and later join them.

This API is not as straight forward as the one in the C# implementations for couple of reasons. Firstly, since it is purely functional, revisions are immutable which requires to create a new revision at each join. Secondly, we need to explicitly create new isolated variables. In the C# implementation the user simply declares which variables have to be isolated and then uses them as any regular variables. The solution in the project implementation is a bit less convenient - it uses special read and write methods, to which the context has to be explicitly stated. However, it makes it much clearer what is actually isolated, when it is accessed and in what context, resulting in a more comprehensive code. Another drawback of my implementation is that it does not allow having isolated from different types in a single revision, a trivial workaround for which is to use tuples or another complex data structure. I considered using a universal OCaml type as an alternative. Universal types in OCaml however can be implemented only using exceptions which are highly inefficient.

Chapter 2

Preparation

2.1 The author in the world of Concurrency

As part of my degree I have gained broad knowledge of the problems that arise from Concurrency and the typical approaches for solving them. The Concurrent and Distributed Systems course gave me most insight into why and how Concurrent revisions can be used for parallel programming. After reading the original paper, the concept naturally fit in and expanded the mental model I have created about the issues and solutions in the world of Concurrency.

2.2 Familiarizing with the OCaml programming language

Prior to starting the project, I had almost no experience with the OCaml programming language. For that reason I dedicated the first part of my project to making myself familiar with it. I used the Real World OCaml book [3] to guide me through the concepts and patterns for the language. I was able to quickly transfer and expand my skills in ML into OCaml without much difficulty.

2.3 The Core and Async libraries

I made extensive use of the Core and Async libraries for OCaml. The latter was used in the core of the implementation and the use cases. The Revision module conforms to the pattern of deferred computations in the Async library. This naturally happened in the development process, since the core concepts in the Async library and those of the project complement each other. The revision and isolated data types are completely isolated, making it trivial to implement them as deferred data types and the forks and the joins as deferred computations.

2.4 Back-up and revision control

For revision control I used git, with which I had a lot of experience. I used GitHub for back-up of the code. To insure myself against hardware failure of my personal laptop, I kept my whole development environment inside a VM. The VM was backed up on an external hard drive so in case of system failure I could quickly continue working on another machine.

Chapter 3

Implementation

I have implemented the Concurrent Revisions concept as a library for OCaml. The implementation has passed a series of unit tests and was used for implementing two use cases - a logging and a chat service, as well as a few simple examples.

3.1 API

Usage of the library is straight forward and effortless. The user first has to satisfy a module signature called `Isolatable`:

```
module type Isolatable = sig
  (** Type to be isolated **)
  type t
  (** Merge function: merge [head] [parent] [current] **)
  val merge: t -> t -> t -> t
end

module Make(X:Isolatable) :
            (Revision with type value = X.t and type isolated = (int * X.t) Deferred.t)
```

Then from this module, using the `Make` functor, he create a `Revision` module that satisfies the following signature:

```
module type Revision = sig
  type i
  type result
  type t
  type isolated
  type value

  val init: unit -> t
  (** Adds a new isolated with [value] and returns a new result **)
  val create:  t -> value -> result

  (** For breaking the result into revision and isolated **)
  val get_revision: result -> t
  val get_isolated: result -> isolated

  (** Scheduling primitives **)
  val fork: t -> (t -> t Deferred.t) -> t Deferred.t
  val join: t -> t -> t

  (** Isolated access **)
  val write: t -> isolated -> value -> t
  val read: t -> isolated -> value Deferred.t

  (** Ensures the revision is determnined **)
  val determine_revision: t -> t

end
```

Creation of revisions is done by the `init` and `create` functions. The former initializes an empty revision and the latter takes a revision and an initial value of the isolated type and returns a `result`, which contains a tuple of `Revision.t` and `isolated`. This tuple can then be broken up by the usage of `get_revision` and `get_isolated`. This seems a bit awkward to use, but it is enforced by the Async library. Both `Revision.t` and `isolated` are deferred types, however since create takes a deferred type as first argument, it has to return a deferred type as well, meaning it cannot return a tuple of deferreds, since in Async each deferred computation has to return a single deferred value.

The scheduling primitives are the trivial `fork` and `join` operations common for asynchronous tasks. Note that they are both purely functional and do not mutate any of the input state and return a fresh revision each time. This makes it explicit whenever the state is changed as this results in creation of a new immutable value. Arguments for the operations over revisions are type checked, which significantly reduces the chance of errors on the part of the programmer. Such errors will be reported by the type checker at compile time. For the illegal revision diagrams when the merge cannot reconcile a valid state, because there is not enough information in the revisions, an exception is raised. There are still some cases when the programmer can implement by error illegal schedules and run them successfully. Ideally these errors should be caught by the type checked or by a more elaborate dynamic runtime check. These are left as a future extension and for now such errors are considered programmer's fault and the behaviour of such is undefined. Undefined behaviour is highly undesirable in a functional environment, especially when doing parallel programming and even more worrying when the aim is to have deterministic behaviour. This issue could potentially be resolved by designing the library as monadic, which is left as a future extension.

Accessing the isolated variables is allowed by the `write` and `read` functions. The former is also purely function and returns a new revision with the updated value. In the case when the isolated is not in this revision an exception will be raised. Again this is not desirable and can be solved by the extension proposed in the previous paragraph.

The `determine_revision` is used to ensure that a revision is evaluated and not merely scheduled for evaluation. Initially the library was not intended to provide such functionality, however during the design of the use cases the need became apparent and I added it to the implementation.

3.2 Revisions behind the curtains - data structures

The internal structure of the revisions is implemented by the following data type:

```
type t = { parent : ((int, Isolated.t, Int.comparator_witness) Map.t);
           self : ((int, Isolated.t, Int.comparator_witness) Map.t);
           written : WrittenSet.t;
           id : int
         } Deferred.t
```

It is a deferred record type which contains two maps that map isolated variables to their value in the parent or the current revision respectfully. Their is also a written set that keeps track of which isolated variables have been updated in order to improve the performance at join time. The `Isolated.t` is itself implemented simply as an integer value that is used as a key into the maps. The `id` field in the revision type keeps track of the id of the last created isolated in that revision chain to ensure uniqueness of ids.

The reasons for choosing these particular data structures are explained in the following section.

3.3 Design decisions

The key aims of the design were to produce a solution that is purely functional, as well as efficient. The desire for a purely functional approach was driven by the necessity of tracking effects of parallel executions. Purely functional implementation ensures that no side effects occur to the isolated variables inside the revisions and ideologically changes happen only when revisions are joined. The benefit of this is that modifications of the state are explicit and easy to track both for the programmer and the run time. Another advantage of this design decision is that replication is very cheap both in terms of time and space, because in OCaml complex data structures which are immutable, such as maps, allocate more memory only when the structure is changed, reusing as much as possible of the internals of the initial structure.

The main design choice for the implementation was how to implement the revisions internally. The obvious choice for the mapping of isolated variables to values was between hash tables and maps. The former would have given constant time for all the update and add operations. However, in OCaml hash tables are usually implemented as mutable data structures. Their imperative nature would have broken the functional model of operation for the library. I could have hidden under the curtains the fact that I am using a mutable data structure by lazy replication. This would mean that upon creating a new revision, the structure representing the parent of this revision (which would be a hash table) would have to be replicated explicitly and values would be added to the current revisions only after writes. This would increase the time and space complexity of forks and joins linearly in terms of the number of isolated variables since each of these operations would have to explicitly replicate the whole hash table containing all isolated values. The functional nature that I was aiming for required creating a new revision after each operation (either fork, join or addition of a new isolated variable). This makes the creation of new revisions the most common action by the library and shifting from constant to linear complexity was highly undesirable. Taking this consideration into account I made the choice of using maps in the internal structure of the revisions. I used the implementation of maps in the Core library. As mentioned in section 1.4.2, they are implemented as AVL trees, which are self-balanced binary search trees, which provide logarithmic complexity for all reads and writes to their structure. Since only insertion and update operations were performed on the maps, all operations on maps we use are of logarithmic time. What is more, since maps in OCaml are purely functional, they reuse a lot of their internal structure and there is no duplication of data.

3.3.1 Complexity

Forking a new revision does not require additional space as the revision structure of the parent is not changed and the forked revision merely points to it. The functional nature of the implementation guarantees that the parent would never be mutated, making this way of replication completely safe. Inside the fork, the only operation that requires extra space is `Revision.write`. It has to change the value of a isolated in a revision. Under the hood this is implemented as a value update in a map. As mentioned previously, this takes just O(log n) time, where n is the number of isolated in that revision and just constant space due to the immutable nature of Core maps and the re-usage of their internal structure.

In similar fashion to forks, joins also create a new revision, heavily reusing the map from the revision to which we are joining. For every variable changed in the joinee, its value in the head revision, the parent revision and the current revision is read, which as already discussed, takes logarithmic time in terms of the number of isolated variables. Then for each written variable, the merge function is called and the result of it is applied to the head revision, creating a new one. This takes time O(k*log n*merge), where merge is the runtime complexity of the merge function, k is the number of variables changed in the joinee and n is the total number of variables in the revision. The join performs k updates to values in the map, which means it has space complexity of O(k + merge), where merge is the space complexity of the merge function. Everything allocated in the merge function is garbage collected after it is executed, which means it does not add extra space complexity to the total runtime. Revisions created by the forks and join would also be garbage collected once they are out of scope, which means space usually is used only for forks not yet joined and the head revisions. We will see how this works in practice with the use cases later on.

In total, a sequence of k forks and joins, n isolated variables, r reads and w writes gives a total runtime overhead of using revisions of O(r*log n + w*log n + k + w*log n) = O((r+w)log n). In most use cases n is a small number, which means that we could regard log n as a constant. This results in unaffected time complexity of the initial algorithm the user is implementing. For example the use cases of a logging and a chat system, discussed later, use only a single isolated. Even when a large number of isolated is needed, we still get only logarithmic overhead. However, using an alternative approach to concurrency would need a more elaborate synchronization scheme that would rely on blocking or roll-backs which in a parallel system would waste a lot of CPU time or cause delays when conflicts arise. This would typically be much greater than the overhead of revisions. What is more, alternative approaches require significantly more effort from the programmer.

In terms of space, the runtime of revisions adds just a constant overhead for each write, which does not affect the space complexity of the algorithm.

3.3.2 Compatibility with existing OCaml systems

All operations on revisions were implemented as deferred computations in the framework of the Async library. This makes the Concurrent revisions easy to integrate in any application using the idioms of the Async library, which is one of the mainstream parallel frameworks for OCaml.

For existing systems build on different concepts, integrating revisions would be clumsy and difficult and would require blocking from inside Async. However this is not a flaw of the implementation itself. Combining concepts relying on different assumptions and providing different guarantees is naturally conflicting and is often undesirable. In fact, this issue is inherited from the Async library, which also is not trivially integrated in legacy OCaml code.

3.3.3 Error handling

There are two classes of runtime errors that the implementation has to deal with.

The first one is concerned with illegal usage of revisions. When a read or write of an isolated from a revision is executed, there is still no guarantee that the revision contains a value for this particular isolated, even when their compatibility was successfully type-checked. In that case an exception `Isolated_Not_Found` is raised. When the run-time discovers an illegal join (note that not all of these are caught), it raises an `Incompatible_join` exception.

The second class of runtime errors are those occurring in the user declared code to be executed inside the revisions framework - namely the merge function and the functions executed inside forks. For the former the exceptions are left uncaught, since merely abandoning joins would result in a silent failure, which would be difficult to debug. As for the latter - the exception is caught inside the fork and only re-raised if the fork is joined. Since results of forks are only applied to the state after the join it is consistent with the concept to treat exceptions as any other state changes.

3.3.4 Unit tests & example code

A number of unit test were designed which cover all possible legal revision diagrams and a different number of isolated variables. They were used in parallel with the ongoing work on the implementation to ensure its correctness.

The simple examples from the original paper were also implemented. These include the previously discussed example of isolating integers, a "Hello World" example using isolated strings and an example demonstrating the benefits of re-raising fork exceptions when revisions are joined, which provides efficient tools for roll-backs of revisions.

3.4 Use case - Logging system

3.5 Use case - Chat server

3.5.1 Motivation

Much like the logging system, a chat system is highly distributed, which has a lot of conflicts that have to be resolved deterministically and requires scalability and limited delays. Such a system can benefit significantly from parallelism, which would allow for more clients to be served and to limit the delays that these clients might experience.

This is clearly a trivial problem, implemented numerous times before. However, it is interesting to see what could be achieved if implemented using Concurrent revisions. I choose this use case since it is exemplary for a problem where we need a lot of concurrency, but without sacrificing consistency.

The aim of the chat system implementation is not be optimized for the particular problem of exchanging personal messages, but rather to be a witness for an arbitrary parallel system. Applying revisions to it and measuring performance between a version with revisions and one less parallel without revisions could give valuable insight on the usability of my library, the performance improvement and the overhead of using revisions. Since the user interface of the client has no connection to the performance inside the server and the qualities I was trying to examine, I paid a little attention on it.

The core component of interest is the chat server itself, since it is the one that has to provide good service to all the clients and the only way to achieve this is to explore parallelism while keeping consistent state.

3.5.2 Features

The following features for usage of the client were implemented for both versions of the server:

- Registration of new users

- Creating, joining and leaving chat rooms
- Sending messages to chat rooms
- Promoting users to admin status in a particular room
- Merging chat rooms

3.5.3 Implementation using Concurrent revisions

I will focus on the implementation of the chat server, since there is much less requirement for performance at the client side and the main benefit of revisions comes in the server implementation.

The chat server has to keep the state used for serving all users consistently. Achieving this in a purely functional manner will require passing the state as a input argument to the function serving the requests. If this function is recursive then this input parameter could be used as a running state. However, this does not provide any parallelism at all. Even when using revisions, we still have to share the head revision between all the parallel serving functions. In such an environment it is almost impossible to explore parallelism in a purely functional manner. For that reason the implementation of the server is not purely functional, but instead has a global reference to the current state. Alternative approach would have been to use monads to circumvent this problem, which are not that common in OCaml and look like a bit of a overkill for the purpose of this implementation.

Data types

The state of the server is of type:

```
type st =
  { id:int;
    rooms: RepRoom.t;
    users: RepUser.t;
    last_event: command;
    last_event_time: Time.t
  }
```

The state keeps track of the last event and its timestamp for synchronization purposes, which I will later explain along with the merge function invoked when joining revisions in the next subchapter.

The two modules `RepRoom` and `RepUser` encapsulate the representation of the ordered set of rooms and users respectively in the state of the server.

Under the hood the types RepRoom.t and RepUser.t are implemented as maps with keys the ids of the users and the rooms and data type chat_room and user. These types have the following signatures:

```
type user =
  { id : int;
    su : bool;
    name : string;
    reader : Reader.t;
    writer : Writer.t;
  }

type chat_room =
  { history : RepMessage.t;
    users : RepUser.t;
    id : int;
  }
```

The record type user has id and name fields for the id and name of the user respectively. A user is also specified by whether he has admin privileges - su, and writer and reader fields that are descriptors used by the Readed and Writer modules in Async. They are used for reading from and writing to the TCP connection to each client in an even-driven manner.

The chat rooms also have ids, along with an a list of users (provided by the RepUser module) and chat history provided by the RepMessage module.

The modules RepMessage, RepUser and RepRoom are used for refactoring the implementation of the server. They all have type t implemented as a map that maps ids (for users and rooms) and timestamps (for messages) to the types representing messages, users and rooms respectively. These modules provide the basic functions required for dealing with ordered sets of these types. Because of the implementation of maps in the Core library, discussed before, all the updates done by these modules take logarithmic time and constant space.

Usage of Concurrent Revisions

As mentioned in the beginning of the previous subsection, the inherited nature of the chat server requires a global mutable state. I implemented this as a reference to a revision with isolation type st (discussed above). There is only a single isolated that is the state of the server in the particular revision. This means that reads and writes of isolated take only constant time and space for reasons explained in chapter 3.2.1.

Unpredictable network delays could reorder events reaching the server. This could be overcome by introducing a partial order to the distributed message passing, which however adds significant complexity and is also out of the scope of this project. For the purpose of this implementation, I consider only determinism based on event timestamps, taken when the events are received at the server.

The most interesting part of the usage of revisions is the merge function. It operates on the current consistent state of the server and the state that has come from the fork. Keeping the timestamp of the last event in the state, allows the merge function to account for some delayed processing of events where they introduce a conflict has to be resolved by incorporating their effects according to their initial timing. Access to the last event from the state is done merely as an optimization, otherwise the merge would have to scan the whole state to find changes instead of only the areas that could have been changed. The reason for this is one of the flaws of the implementation of the library - ideally we would like the user list and each room to be isolated separately, however my implementation do not allow a revision to have isolated of different types, for reasons explained in chapter 3.2. Given that it was not too difficult to deal with that limitation I do not consider it a major issue. The merge function also has to account for many possible sequences of events that could introduce conflict which have to be resolved deterministically in order to keep the deterministic change of the state. Let's take a look at some of these special cases:

Consider the case when two users simultaneously try to register with the server with the same name. A new revision is forked for each of the registration events. The function evaluated in the fork has a consistent snapshot of the state at the time of the fork, but is not aware of what the happens in the other fork. Given the name the users try to register with is not already in use, both forks will succeed resulting in two new revisions ready to be joined to the global state. This introduces a conflict - having two users with the same name in the two revisions I have to join. This is where the merge function I supplied when isolating the state shines. When joining the first revision everything works fine and the result of the join is assigned to the global state. When joining the other revisions however, it is joined to the new global state which already has the first user registered. The merge function can how look at the list of users in the head revision and find out the user name is already in use, inform the user who tries to register and abort the join without introducing inconsistency in the state of the server. Unfortunately there is still a bit of non-determinism - the user that succeeds is not necessary the one that tried to register first. Alternatively the merge function could take into account the timestamp of each registration event and ensuring the first user is the one registered. This would mean that the second user could potentially be registered and later on thrown out of the server, which is an undesirable poor user experience. The same reordering can happen because of network delays as well. In my view this is not a flaw of the concept or the implementation but rather a flexibility that allows introducing a moderate amount of non-determinism, when determinism is not needed or too expensive.

Another example of a special case is when a new user enters a chat room and another user sends a message to the same room after that, the effect of the second event is joined first to the state. The merge function can easily deal with that as well - it can simply compare the timestamp of the registration with the timestamps of the messages in the room history and send the last message to the newly registered user.

The final example of special case with which the merge function has to deal with is merging two rooms. This means that that messages from both groups have to be put in the right order and the union of lists of users has to be taken. This can trivially be done inside the fork, since both the history and the user lists are implemented as maps, taking their union is trivial and efficient. However before the result of that is joined to the head revisions, some other users and messages could have been added to one of the rooms. The merge function once again deals with that by checking the timestamp of the merge event to make sure all new messages are added accordingly and checks once again that a user is in the resulting user list if and only if it is in one of the groups at the head revision.

Runtime flow

Each user that wants to communicate with the server opens a TCP connection with it. Then the server and the clients exchange commands that are predefined and are shared between them. They are passed around serialized as s-expressions. S-expressions are an efficient serialization technique in OCaml, based on well bracketed expressions.

All the connections are served in parallel and commands are added to a pipe as they arrive without blocking. A recursive function takes commands out of this pipe and forks a new revision corresponding to each command. The forked revisions are just scheduled for evaluation without blocking the function that takes commands out of the pipe, which means that at any given time there can be arbitrary many fully parallelized forks.

Again in full parallel, the revisions of that result from the forks are added to another pipe after they have been determined. Another recursive function takes revisions (which are already determined) and joins them one by one to the global state. Note that this function will never block waiting for a revision coming from the pipe to be determined as all revisions in the pipe are guaranteed to already have been evaluated. Parallelizing the joins is not possible as this will result in invalid revision diagram, which diverges forever without even converging to a globally consistent state. Having said that, the run-time of the server is as concurrent as the concurrent revisions concept allows for.

Updating a reference at the end of the join, which is also dereferenced at the forking time and in the beginning of the join, is safe since all deferred computations in Async are atomic and guarantee that consistent global state is always seen both by the forks and the joins.

3.6 Remaining issues

Here I will summarize the remaining issues with the implementation of the library, discussed previously, and propose solutions for them.

The most important issue is the fact that the user can implement and successfully run some illegal revision diagrams. This is clearly undesirable, since such schedules are undefined which introduces a non-determinism and creates a large room for programmer errors. Some of these at the moment are caught by either the type checker or by run-time checks. However this is clearly not enough. Ideally all illegal schedules should not be allowed by the type system. This can be done by using monadic design for the whole implementation. Since it does not affect the evaluation of usability and performance of the concept implemented in OCaml it is left as a future extension and such errors are considered programmer's fault and invoke undefined behaviour.

Another important issue is the lack of support for isolating more than one type in the same revision. In most cases this can be circumvented by using complex data types and accounting for the different types inside the complex data structure in the merge function. However the ability to have different isolated types in the same revision would allow for more fine grained merge functions that will improve the control over conflict resolution and reasoning over the logic. This also can be resolved by monadic implementation. As the lack of this feature does not cripple the usability and can be worked around easily, it is left as a future extension.

Chapter 4

Evaluation

4.1 Fitness of the concept for the use cases

Was it easy to implement them in that way? Did it make me thing in a different way (was it better)? What was exceptionally good about it?

4.2 Negative aspects

Any syntactic of conceptual awkwardness encountered along the way

4.3 Performance evaluation

4.3.1 Experimental data

Plots, analyzes, etc.

4.3.2 Performance analyzes

Time, space, what seem to be a bottle neck, were the design choices right?

Chapter 5

Conclusion

It works! (or not)

Chapter 6

Bibliography

[1] *Concurrent Programming with Revisions and Isolation Types*, Sebastian Burckhardt, Alexandro Baldassion, and Daan Leijen. OOPSLA'10

[2] *Source repository:* https://github.com/dpp23/ocaml_revisions

[3] *Real World OCaml* Jason Hickey, Anil Madhavapeddy, and Yaron Minsky; O'Reilly 2013

[4] *Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987* P.A.Bernstein, V.Hadzilacos, and N.Goodman.

Appendix A

Project Proposal