Dimitar Popov

Concurrent revisions library for OCaml

Part II of the Computer Science Tripos

Homerton College

February 1, 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 Concurrent revisions

The idea of Concurrent revisions was initially proposed at OOPSLA'10 [1]. It highlight 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 allow the typical functionality for asynchronous tasks - the user can create, fork and join them.

1.2 Motivation

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.

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.

This approach is data centric in a sense that it takes the complexity of synchronization out of the tasks and adds it to the data declarations.

1.2.1 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[reference].
- **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 and logic, write-back 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.
- **Chat systems** - The usage of functional languages for large commercial systems is increasing. One example of that is the Facebook chat, which is written in Erlang. It is a large distributed system that has a lot of conflicts, timing and consistency are vital and it more or less requires 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 a chat server using them.

1.3 Deeper look into the concept

1.3.1 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 simple example:

```
1  IntIsolated = isolate(int)
2  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 fork two new revisions that would credit the account with 5 and 10 pounds respectively. At this point `account` has different value in each of the three revisions.

Then we join the two new revisions one by one to our main initial revision (line 13 & 14). Luckily due to how we specified the merge function and the deterministic nature of the approach, we 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, while now we simply synchronize it when we join the tasks.

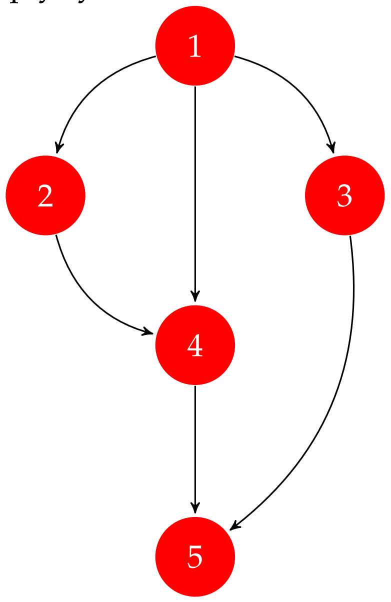

Fig.1: Revision diagram of Example 1. In the following diagram the nodes are the revisions. Outgoing arrows represent forks and joins are represented by incoming arrows, which are always two, one straight for the revision we are joining to and one bend for the joinee. In the diagram nodes correspond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

1.3.2 Illegal revision diagrams

Not all possible joins are legal as some of them might invalidate the assumptions about the flow of revisions. Consider these examples of illegal diagrams:

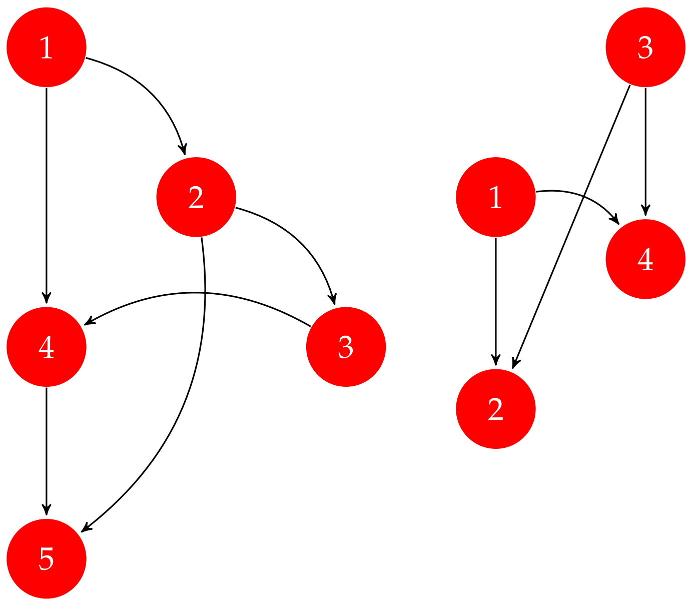

Fig.2: Illegal revision diagrams.

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 as we are interleaving the joins and we might merge the work done in 2 effectively twice.

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.

1.3.3 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 a illegal join is performed. 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:

```
1  module IntRevision = Make(struct
2      type t = int
3      let merge head parent current = head + current - parent
4    end)
5
6  let () =
7    let r = IntRevision.init () in
8    let res1 = IntRevision.create r 0 in
9    let revision = IntRevision.get_revision res1
10   and account = IntRevision.get_isolated res1 in
11      Deferred.both (IntRevision.fork revision
12                      (fun r -> return (IntRevision.write r account ((IntRevision.read r account) +
13                    (IntRevision.fork revision
14                      (fun r -> return (IntRevision.write r account ((IntRevision.read r account) +
15      >>|(fun (rev1, rev2 ->
16         let join_rev1 = IntRevision.join revision rev1 in
17         let join_rev2 = Intrevision.join join_rev1 rev2 in
18           assert(IntRevision.read join_rev2 account = 15)
```

Example 2.

In Example 2 we can see the actual implementation of the pseudo code in Example 1. From line 1 to 4 the user creates the `IntRevision` module using a simple anonymous module specifying the type of the isolated data and the merge function. Then on line 7 he initializes an empty revision and one 8 and 9 he adds a new isolated to the initial revision to create a new one. He them forks the two asynchronous account credits (line 11 to 14) and later joins 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. What is more, it does not allow having isolated from different types in a revision, a trivial workaround for which is to use tuples.

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, mainly because the library enforces them. Another reason for this is that 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.

Chapter 3

Implementation

3.1 Nuts and Bolts

In depth description of the internal structure, the API.

3.2 Design decisions and trade-offs

Quick creation of revisions (also space consuming) vs quick read/write. Chose the later, because it looked as a better fit for the use cases, but the other one is equally valid solution. Othes design decisions

3.3 Use cases

3.3.1 Chat server

(not yet implemented)

why it is usefull to do it that way (trivial problem, but doing it in this framework is interesting and could show much) structure & implementation (briefly) experimental data - a lot of chat rooms, merging chatrooms, conflicts; some plots could be good sequential version (hopefully) compare both

3.3.2 Other one?

3.4 Remaining issues

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

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

Appendix A

Project Proposal