Dorian Peake

Parallelism in OCaml Under the JVM

Computer Science Tripos

St John's College

April 28, 2014

Proforma

Name:	**Dorian Peake**
College:	**St John's College**
Project Title:	**Parallelism under OCaml using the JVM**
Examination:	**Computer Science Tripos, July 2014**
Word Count:	**TBC**
Project Originator:	Dorian Peake
Supervisor:	Jeremy Yallop

Original Aims of the Project

The aim of this project was to highlight the differences between single and multithreaded OCaml code. This was done by adapting the Lightweight Threading Library (LWT) – a very popular threading library for OCaml – to use the OCaml to Java bytecode compiler OCaml-Java, thus allowing software written with LWT to utilise the multithreading capabilities of the JVM.

fill this out a bit more, maybe talk about what differences the project was looking for/found

Work Completed

Pinkbook says at most 100 words summarising work completed, e.g. scheduler of LWT to OCaml-Java threads, worker public/private queue structure, benchmarks, etc. The LWT threading capabilities have been integrated with OCaml-Java, thus allowing some LWT programs to compile to Java bytecode and run in parallel on the JVM. This is done using a scheduler algorithm which maps LWT threads to Java threads.

fill this out some more as well when

Special Difficulties

Pinkbook says at most 100 words describing special difficulties, but most should just say "None" None, *possibly.*

Declaration

I, Dorian Peake of St John's 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 [signature]

Date [date]

Contents

List of Figures

Acknowledgements

add thanks at the end

This document owes much to an earlier version written by Simon Moore . His help, encouragement and advice was greatly appreciated.

Chapter 1

Introduction

1.1 Motivation

Trends current trends in chip advancement have shown that since around 2003 [11], the growth in processor frequency has effectively plateaued and computer architects have had to look for different methods of achieving greater computational power. The diminishing returns of processor frequency have been greatly effected by the amount of energy required to switch a transistor's output – known as the *dynamic energy*. This dynamic energy needs to be dissapated through the side of the chip die at a size of around 1.5cm. Clearly this becomes an increasingly difficult task for processors with greater clock frequencies which generate more heat energy to dissipate. As a result of this and some other unmentioned factors, the focus of achieving greater performance has moved from clock frequency improvements to parallel processing. Therefore it is important for software developers to create software that is able to take full advantage of the hardware of today and estimate ways to make it scale on the hardware of tomorrow.

this para is sounds like a history talk which is what thingy-bob told us not to do

Functional Programming concepts have been around since Turing and Church's work in the 1930's [14] but have become more popular in recent times. The expressiveness that functional programming paradigms give the programmer has finally caught on in mainstream development and as such languages such as C++ and Java are incorperating notable functional programming techniques such as type inference [6] and first-class anonymous functions (lambas) [7]. Notable functional languages include Haskell and ML which have inspired more widely used functional languages such as OCaml [13]. OCaml is a very extensive functional language which also includes an elegant object system, thus incorperating the best of both functional and imperative programming techniques and making it a very powerful language to develop with. Unfortunately OCaml in it's current vanilla state is unable to compile parallel code, which is becoming an ever more neccessary approach to development as systems become more and more parallel.

cite where it says ATM's and credit card chips use Java

find where this slogan is written

Java on the other hand is a very good imperative programming language/system. It compiles to bytecode which runs on it's virtual machine which is implemented in everything from ATM's and credit card Chipsto home desktops and servers – leading to one of the slogan's '*write once, run everywhere*'. Over the years the JVM has become highly optimised to the hardware/systems that it runs on such that there is little diffrence between Java code running on the JVM and natively compiled code. Furthermore, Java supports parallel processing, fully utilising any number of cores/hardware threads that the system has to offer.

A peice of software called OCaml-Java aims to merge best aspects of both languages together to create a better OCaml programming solution. It achieves this by compiling OCaml code to Java bytecode, thus allowing it to run on any JVM. This means two things:

- OCaml can now be written to run on a wide variety of hardware/systems.
- OCaml can use Java's multithreading capabilities to run fully parallel OCaml code [4]

OCaml-Java comes with a basic collection of concurrency tools which are essentially a small subset of Java's own concurrency library, with wrappers which allows access to them from within OCaml. Clearly the concurrency techniques provided by Java do not nicely translate to the way concurrency is done in a functional programming language like OCaml. There already exist extremely good and highly optimised OCaml concurrency libraries such as Async by Jane Street and LWT (Lightweight Threads Library) by Ocsigen, however these will not directly work with OCaml-Java when compiled to Java bytecode. Somehow getting one of these OCaml concurrency libraries working with OCaml-Java would demonstrate OCaml-Java's true potential for parallel OCaml code, written in the functional flavour that works best with the OCaml language – this is the motivation for my project.

My project aims to interface OCaml-Java and the LWT concurrency library in order to demonstrate OCaml-Java's potential for parallelised OCaml code. The rest of this document outlines the preparation and research I conducted in order to plan my approach to this project in chapter 2; the methods and processes that were used to implement the project in chapter 3 and the result of the project development and the work process as a whole in chapter 4. My code can also be found in chapter 7.

1.2 Related Work

Chapter 2

Preparation

The following chapter outlines the processes taken in preparation for the project's implementation. I begin with a look into OCaml's multithreaded capabilities, including the limitations on developing parallel code and why these limitations exist. I also discuss LWT's own asynchronous library in comparisson to Java's and the difficulties that may occur when attempting to interface the two. OCaml-Java is overviewed, including what tools become available with it's use and how it manages to provide the user with concurrency when writing parallel OCaml code. Finally the research is concluded with a set of requirements which I believe are representative of what the project should hope to achieve and the tools/software I will be using throught it's development.

2.1 Investigating Multithreaded OCaml

In order to outline the best implementation plan for the LWT–OCaml-Java interface, it was neccessary to understand the limitations that OCaml possessed in compiling parallel code. The OCaml-Java paper from the 2012 'Trends in Functional Programming' collection [4] gives rise to the reason – stating that the Core OCaml library is not reentant and the OCaml garbage collector is neither parallel nor concurrent. Making the Core library reentrant is a relatively straight forward change to the system, however making the garbage collector parallel is not as simple. Ultimately there are many ways to perform concurrent and parallel garbage collection, however this has proven to be 'too complex' for the current state of the OCaml system. Java's G1 is fully paralleland allows multithreaded support of Java applications, therefore by compiling OCaml code do java bytecode, the language may utilise Java's parallelism capabilities.

cite this, definitely saw this some-where

cite me

It was neccessary to decide which of the popular concurrency libraries available to OCaml I should use for the purposes of this project therefore I researched into OCaml's ability to produce concurrent OCaml code, which lead me to OCaml's own inbuilt threads libarary. The OCaml manual [12] describes the OCaml threads as being concurrently on a single processor using time sharing. However there exist other concurrency libraries which boast a richer feature set than the one provided by OCaml as default. Examples of these languages are Async – developed by Jane Street Capital– and LWT (Lightweight threads library) – developed by Ocsigen. These are the two main concurrency packages used within OCaml development, choosing one to use within my project was simply a matter of popularity, my reasoning is that they are both extremely good packages and would serve well to demonstrate the capabilities of multithreaded OCaml-Java.

cite me

cite me also

2.2 Investigating LWT

LWT is an extensive concurrency suite for OCaml which is based around the idea of lightweight cooperative threads [10] where cooperative threads manage their own lifecycle as opposed to preemtive threads which are controlled externally, e.g. by the operating system. Creating and destroying a thread in LWT are very cheap operations such that the system will create a thread for every system call that is executed. This idea of lightweight threads with a small lifetime handles itself well to the way OCaml (and languages within the ML family) work which is allocating new data frequently which have a small lifetime. The OCaml garbage collector is also optimised for this by using generational garbage collection techniques to more frequently collect younger data, as data that has survived a garbage collection once or more before is likely to have an extended lifetime. [2]

On the other hand, Java threads have a greater creation overhead and as such cannot be used in a similar fashion to the way LWT threads work without detrimental effects to performance. Java's java.util.concurrent implementations avoid the creation overheads by utilising thread pools [9]. This observation lead me to the implementation decision of creating a thread scheduler which maps LWT threads to Java threads as clearly a 1:1 mapping would simply expose great amounts of thread creation overhead, possibly crippling any speedup gains would obtain.

There are many functions within LWT to create threads which have various different effects. One of the most commonly used is the *bind* function, with the following type:

```
val bind : 'a t -> ('a -> 'b t) -> 'b t
```

Essentially the bind function takes a thread of generic type, a function which takes the return value of the thread, performs some processing returns a new thread. This processing connects the passed thread and function such that when the thread has finished executing, the function will be run with the return value from the thread. This acts synonymously to the classic 'sync' operation within various writings on parallel processing [5]. The new thread returned by the bind function is the result of the whole bind operation – that is, once the passed thread has executed and the passed function has returned some value.

Since threds in LWT are cooperative, theres no real notion of spawning, starting or stoping such as in POSIX or Java. Asynchronous execution is achieved by binding some work to a thread which will execute later on. This can be achieved with the 'Lwt_main.yeild' function:

```
val yield : unit -> unit Lwt.t
```

The *yield* function returns a thread which simply goes to sleep, then wakes up as soon as possible and terminates. This is entirely useless on it's own but binding to this thread will create a situation where you're able to schedule something to run in the near future – without knowing when it will occur.

2.3 Investigating OCaml-Java

Providing the framework to parallel OCaml code in this project, OCaml-Java is a collection of Java programs and libraries that allow you to compile OCaml code to Java bytecode for execution on the JVM; run OCaml bytecode compiled software on the JVM using an interpreter written in Java; and access Java libraries directly from within OCaml code using a specially designed interface that maps Java classes to OCaml types. However most relevant to this project is OCaml-Java's ability to run OCaml code in parallel using Java threads.

OCaml-Java provides a concurrency module which contains many standard concurrency features such as Locks, atomic variables and condition variables. The locks, for example, can be manipulated as simply as calling 'make_reentrant' to construct a new lock and 'lock' and 'unlock' to change acquire/release the lock respectively:

```
val make_reentrant : bool -> t

val lock : t -> unit

val unlock : t -> unit
```

Essentially, the concurrency module is just a set of OCaml wrappers around their Java counterparts [3] therefore the OCaml-Java reentrant lock module 'Concurrent.Lock' is a wrapper around the Java reentrant lock class 'java.util.concurrent.locks.ReentrantLock'.

For classes that do not yet have an OCaml wrapper, there is a neat interface which allows direct calls to Java functions and manipulation of Java references, for example object instantiation. Via the Java.make and Java.call functions, you are able to instantiate any Java class or call any Java function:

```
let obj = Java.make "java.lang.Object()"
let itg = Java.make "java.lang.Integer(int)" 1231

let obj_hash = Java.call "java.lang.Object.hashCode():int" obj
let eq = Java.call "java.lang.Object.equals(java.lang.Object):boolean" obj itg
```

The code above gives an example of using the Java.make and Java.call functions. Two Java objects are created named 'obj' and 'itg' using different classes via the Java.make function, then their hash values are compared to each other using the Java.call on the hashCode method and the equals method. Whilst the java primitives are easily mapped to OCaml types e.g. a Java int maps directly to an OCaml int32, the types of Java objects need to have their own defining types within OCaml. The typing scheme introduced makes a java Object map to an OCaml type ''a java_instance' where ''a' is a Java class. Classes are denoted within OCaml by substituting dots with quotes to abide by the OCaml syntactic rules [3], therefore the equivalent OCaml type for 'java.lang.Object' would be 'java'lang'Object java_instance'.

2.4 Requirements Analysis

As a result of my investigations into OCaml, OCaml-Java and LWT, I belive that the interface between multithreaded OCaml-Java and LWT must meet the following requirements:

Requirement 1 As discovered in section 2.2 OCaml threads and Java threads behave very differently, especially in terms of thread creation overheads. Therefore it is neccessary to create a mapping from LWT threads to Java threads using a scheduler to perform the mapping operations.

Requirement 2 ??

Requirement 3 ??

2.5 Choice of Tools

My project depends entirely on the use of OCaml and Java therefore there is no real decision relating to the programming languages I will use. Furthermore LWT and OCaml-Java are two libraries that I will be depending on throughout the development of the project. These invariants aside, the following section outlines my choice of development environment, including version control, backup precautions taken/software used and the development and build methods applied.

2.5.1 Development Environment

Software and Systems

Unlike Java, OCaml has little in the way of a fully supported IDE. There are addons for IDE's such as Eclipse[1] which add syntax highlighting and build functions, however most (including my supervisors) use a text editor such as Emacs or Vim to develop – which both come stock with OCaml syntax highlighting. I am already very familiar with Vim, therefore it was a clear choice to use it. I also used Mac OSX as my host operating system as I am a mac user, however at since there are little restrictions to what operating oystem OCaml (and in particular Java) can run on therefore I was also able to utilise the Intel Labs PWF machines on occasion which are running Ubuntu version something.

find pwf facilities version

[1] https://www.eclipse.org

Version Control

For version control I used Git since I'm already familiar with it and it has always provided me with the neccessary tools I need to work effectively. I also used Github to host my code in the cloud for a few reasons:

- Github provides educational users with free private repositories which I used for this project.
- Storing my code on the cloud means that I can easily access it from anywhere with internet access, with all my revisions.
- My supervisors are also on Github meaning they are easily able to check my progress.
- Github's interface is appealing to use

remove this

Backup

In conjunction with Github as my primary backup solution, I also hosted my code on Dropbox in the case I loose work I haven't yet committed. Furthermore I also have mirrors of my Git repository hosted on the PWF facilities which were pushed to frequently.

2.6 Summary

Within this chapter I have demonstrated the process of planning and investigation taken in order to prepare for the projects implementation. I have investigated the ways that OCaml and LWT are able to perform asynchronous code execution, including the cause for OCaml's limitations in fully parallel code compilation and the differences between LWT and Java's handling of threads. Furthermore I have demonstrated a few tools that OCaml-Java has introduced which allow parallel OCaml code to be written and run on the JVM. Using this information I constructed a set of requirements that I believe are neccessary for this project to adhere to in order to achieve the success criteria. Finally I explained that Vim will be my choice of IDE, running on Mac OSX; Git and Github will be used for version control and in addition the PWF facilities and Dropbox will be used for backup.

Chapter 3

Implementation

3.1 Parallelism Overview

The structure that LWT follows in order to create an asynchronous framework ultimately determined what the neccessary steps were for me in implementing the LWT – OCaml-Java interface. LWT achieves asynchronous execution by storing yielded threads (mentioned in section 2.2) in an queue which will be referred to as the 'yield queue' from this point onwards. The function Lwt_main.run runs an algorithm which takes an initial thread and continuously polls this passed thread for completion (at which point b value has been returned). if the thread has not yet returned a value, the algorithm iterates through the yield queue, executing all the threads contained within it until it reaches the top of the queue. The algorithm then re-polls the passed thread for completion again, and cycles through the process of executing threads on the yield queue and polling for completion until a value is returned.

draw a diagram of thread queue being executed

The functions to be executed by the yield threads will most likely add more threads to the yield queue. These threads are added to the bottom of the queue and are not executed until the next polling iteration. Figure blabladescribes the functionality of the yield queue graphically. My approach to parallelising this model was to mimic the functionality of the Lwt_main.run function accross many Java threads using OCaml-Java. These OCaml-Java threads are known as 'worker threads', which perform the process of taking threads off the yield queue and executing them. Listing blablablasummarises the behaviour of a worker thread. Initially the OCaml thread (that is, the first thread executing the main OCaml code) calls Lwt_main.run with an initial (passed) thread to execute – just as normal. However the function now spawns multiple worker threads (dependant on the number of physical threads the system has available) which are

find out how to reference listings

set to continually run LWT threads on the yield queue until interrupted. The OCaml thread begins work on the passed thread which will inevitably spawn other threads to be pushed onto the yield queue. Once the initial thread has finished work and is waiting on the spawned threads to complete, the OCaml thread will continuously poll the passed thread for a return value. Once this return value has been received, the OCaml thread sends an interrupt signal to all the other worker threads in order to stop processing yield threads and terminate execution. It is interesting to note that the OCaml thread is governing the other Java threads in a preemptive style whereas (as mentioned in in section 2.2), LWT threads are cooperative – there were no problems working with the two threading ideas together.

As an example, listing blablablais code that when run, simply outputs what thread it is running on. It does this 20 times before quitting, leaving a trace of all the threads that were executing at that time. The inital passed thread is executed which spawns a yield thread that is placed onto the yield queue, furthermore another function (the waiter function) is bound to the yield thread such that when the thread is awoken and terminates, the function that was bound is then called with the return value of the yield thread. This function simply prints out what thread is currently running this particular code and then adds another thread to the yield queue, making itself the waiter function. This causes a continuous cycle where threads are added to the yield queue, taken off by some worker thread, and the name of the worker thread is printed out. There is a race between the worker threads as to which one picks up the yield thread, therefore the output from running this program is different each time, demonstrating real parallel asynchonous execution.

insert that code i wrote just now in here soon

With the overview of the idea for implementing parallel thread execution in LWT, my approach took two main paths: a naive approach of using a runtime lock and a coarse grained locking approach. I decided upon these two strategies because considered the first approach to be a relatively straight forward way of getting a result from the project whilst also giving insight into potential problems. The second approach was partly based on the concurrency options that OCaml-Java provided for use: OCaml-Java provides reentrant locks and semaphores amongst the atomic variables within the concurrency module, although I had initally decided to use semaphores with integer values based around the number of threads currently on the queue, as long as there is code running, there will be threads on the yield queue because every call in LWT results in a thread. Implementing semaphores would cause more overheads with little returns, therefore a simple locking scheme was a more sensible idea in this respect.

3.1.1 Naive Runtime Lock Approach

draw a picture of the naive runtime lock situation

This approach involved having all threads require the 'global runtime lock' before attempting to pull anything off the yield queue. The lock is also held all whilst the thread is being processed and is only released when all processing has been finished and the worker thread is about to enter its next iteration of work. All other threads are blocked, waiting on the runtime lock. Figure blablabladescribes this situation graphically. This method, although running in truly in parallel (i.e. many threads performing processing at a time), loses any performance gains from it – in fact it has introduced the overhead of now requiring a lock to be held before every thread is executed. However this approach has created a foundation for a more efficient method to be implemented.

3.1.2 Coarse Grained Locking Approach

'full' or 'more'?

Sustituting a global runtime lock for a coarse grained locking implementation allows the project to take full advantageadvantage of the parallelism exposed by OCaml-Java. The aim of the coarse grained lock is to achieve the following requirements over the global runtime lock:

1. The locks should prevent unneccessary blocking of threads.
2. The locks should allow atomic access to shared and mutable data
3. The locks should do something more

rule of three, need one more here

Threads in LWT have mutable state, consider a thread state going from 'Sleep' with no data to 'Return' which lets anything accessing the thread that the thread has finished executing and has returned its value. It is neccessary for the old 'Sleep' thread state to be removed (and collected by the garbage collector) and the 'Return' state to be created so the value can be accessed. Therefore the locking mechanism requires that only one worker thread may be accessing a state at a time to ensure atomic access to the mutable state data (so as to conform to requirement 1). Furthermore, since each thread may have only one state, the best decision was to associate a lock with each thread which is to be acquired by all functions that access the thread.

Threads in LWT take a further unusual form of 'thread_repr' which are thread indirection types. They provide mutable references to threads (which OCaml does provide, however this is a wrapper around this functionality) – but ultimately it serves as a method of performing type abstraction to ensure all variations of threads can be thought of as some form of 'thread representation/thread_repr'.

An interesting result of this is that there may be many layers of redirection within a reference to a thread – as depicted by figure something. Therefore it is important to ensure the behaviour of a thread lock is consistent and sensible when dealing with this layering. In deciding what the best approach for applying the thread locks in this situation, I had two methods to choose between:

draw figure of layers of thread_repr's

1. Apply a different lock at each individual level, ensuring all locks are acquired and released at all levels below the current 'thread_repr' level before manipulating the thread data.

2. Have a lock at the lowest level only and ensure this is shared correctly between all other layers.

I decided to go with the second option and share the lock between many levels of the thread_repr chain. This seemed like the cleanest option, adding little overhead to the locking process and moving it to the slightly more infrequent thread creation functions.

Thread Safety

Half-Baked Thread Problem Some aspects of how LWT handles threads did not directly map well to a parallel implementation of the library. In some cases this meant that special care was neccessarily taken to handle events in the correct way. One notable example of this is a race condition between threads being placed on and taken off the yield queue before being appropriately initialised.

A visual description of the problem is outline in figure somethingalong with the offending code segment in listing something else. The problem occurs when attempting to bind a thread to a yield thread, so that it may be run asynchronously. The yield function adds a thread onto the yield queue instantly without the waiters being added to a waiter list. This means that any worker thread is able to remove the yield thread from the queue and run it. At the point the bind has added a waiter function to the waiter list, the yield thread has finished executing and will therefore never run the waiters now attached. Finally the garbage collector picks up the yield thread which has finished and it is never to be seen again.

draw best representation of this problem as possible

add bind code somewhere here

Although the gap between where this race condition occurs within the code is relatively small, the problem occurs suprisingly frequently. To overcome this issue, it was neccessary to lock the yield thread between the point of creation and the end of the bind. However this fix makes an assumption on how the yield thread is used – it suggests that a yield thread will only ever be used in conjunction with a call to bind. While this isn't strictly true, since a yield thread is a fully qualified thread on its own that simply does nothing, justification to this assumption is that – to my understanding – it is only ever sensibly used in conjunction with bind. To create a yield thread which sleeps and wakes with no functions to run upon termination is – useless. Furthermore the only way to achieve some kind of useful processing with a yeild thread is to attach a waiter function which will run when the thread terminates.

To lock and unlock the yield thread separated accross two functions, I used an additional thread value containing an option for a bind callback. A thread using bind which requires post bind processing can fill this option value with a function that performs the appropriate post-processing. Bind, upon completing its binding operation, will then execute the callback if it exists, finishing off the thread processing appropriately. Using this new bind callback, the call to yield first locks the yield thread before creation and adds a bind callback function which unlocks the thread. In this state, no worker threads will remove a thread on the yield queue which is currently locked (a requirement for removing a thread from the yield queue is that the worker thread is able to acquire the thread lock). Once the bind operation has completed on the yield thread, the callback is run which unlocks the thread, allowing it to be picked up by a worker thread to begin execution. A visual description of this change is outlined in figure something

draw new diagram of locked thread being bound and then unlocked later on, possibly (seems hard to draw).

ACID Callbacks Callbacks in parallel processing can be called by any and all threads that have access to the code and shared data at any time, therefore it was also important to the rewrite the LWT callbacks such that each thread accessing the callback does so in isolation to any other concurrent thread, the shared data is accessed atomically, the shared data remains in a consistent state and the shared data changes remain – which are the so-called ACID properties of database transactions. For two out of for of the ACID properties (consistency and durability) LWT's code already manages well, however atomicity and isolation issues needed to be dealt with. This mainly consisted of converting shared data to their atomic type equivalent, and ensuring data dependencies were kept serialised.

3.2 Backing out of I/O

3.3 Profiling

Chapter 4

Evaluation

Chapter 5

Conclusions

Chapter 6

Bibliography

Bibliography

[1] Jane Street Capital. Technology - Jane Street. http://janestreet.com/technology/.

[2] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications with Objective Caml*. O'Reilly, 2000.

[3] Xavier Clerc. Ocaml-java 2.0. http://ocamljava.x9c.fr/preview/.

[4] Xavier Clerc. OCaml-Java: OCaml on the JVM. *Trends in Functional Programming*, pages 167–181, 2012.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2013.

[6] Microsoft Corperation. auto Keyword (Type Deduction). http://msdn.microsoft.com/en-us/library/vstudio/dd293667.aspx.

[7] Oracle Corperation. Lambda Expressions. http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

[8] Oracle Corperation. Learn About Java Technology. http://www.java.com/en/about/.

[9] Oracle Corperation. Thread Pools. http://docs.oracle.com/javase/tutorial/essential/concurrency/poo

[10] Jrmie Dimino. Lwt User Manual. 2012.

[11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantative Approach*. Morgan Kaufmann, fifth edition, 2012.

[12] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rmy, , and Jrme Vouillon. The OCaml System Release 4.01. 2013.

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

[14] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, third edition, 2009.

Chapter 7

Appendices

Chapter 8

Project Proposal

8.1 Introduction

The concept of functional programming is becoming far more widespread and important in recent times. Object orientated and imperative languages like C++ and Java are beginning to use functional concepts such as type inference, first-class functions and more. The C++11 specification defines a new 'auto' type which causes the compiler to infer the variable type from its content [6]. Furthermore, Java 1.8 now includes Lambda Expressions in conjunction with a range of modifications to the libraries that allow its use [7]. Companies such as Jane Street – a quantitative trading firm – use OCaml entirely for all their high performance trading software and algorithms [1], and contribute to the advancement of the OCaml language.

OCaml is a great example of a powerful and robust functional language. It includes everything you would expect from a functional language and more such as object orientated and imperative programming paradigms, and is derived from the highly expressive ML programming language. Its powerful type system provides great type safety, eliminating many of the associated runtime errors; automatic memory management through it's incremental garbage collector and strict evaluation branching from it's ML roots in theorem proving. Parallelism in OCaml is however limited due to a global runtime lock present in the system. The purpose of this lock is to prevent unsafe use of non re-entrant code within the core OCaml library, but more importantly is present due to the fact that the OCaml garbage collector is not multi-core friendly [4]. The resulting issue is that only one piece of OCaml code may be executed at a time leading to very little capacity for parallelism.

Java on the other hand is a widely supported, widely available software platform and programming language used in devices like desktop/laptop PC's, mobile phones, ATM's and even credit/debit cards [8]. Its huge community of developers, wide selection of libraries and ability to write easily portable and distributable software means it has continued to grow rapidly over the years. Java also has a thread safe concurrency library which works in conjunction with its parallel and concurrent G1 garbage collector.

Both languages have their pros and cons however the OCaml-Java project aims to bring the best of both languages together by compiling OCaml to Java bytecode, taking advantage of OCaml's powerful typing system and Java's feature rich library. In addition, OCaml code compiled with OCaml-Java uses Java's G1 garbage collector meaning that the software is able to run in parallel across multiple cores without restriction. OCaml-Java comes with a relatively low-level concurrency library which allows developers to utilise its multi-core capabilities, however these low-level modules can be cumbersome to use frequently, especially considering there are some very popular and powerful concurrency libraries for OCaml – such as Ocsigen's Light Weight Threads (LWT).

8.2 Substance and Structure of the Project

The aim of this project is to port the LWT concurrency library to use OCaml-Java's low-level threading library therefore taking advantage of Java's multi-core processing capabilities and providing an easier, more robust means of writing parallelised OCaml code to run on the JVM. LWT is a widely used monadic concurrency library for OCaml with strong emphasis on the 'light' aspect of their threads. Spawning a thread in LWT is a very cheap operation and as such, threading with LWT fits well with the short-lived-data aspect that is well acquainted with functional languages. Furthermore the results of the project will provide some interesting results into the increased scalability of OCaml code running in parallel across multiple cores.
Thinking modularly, the project can be divided into three distinct sections:

Porting the LWT Library: Porting the library to use OCaml-Java will allow the execution of software written in LWT under OCaml-Java, running on a single thread. This will mainly consist of writing a back-end to act as a pipeline between LWT and OCaml-Java.

Making LWT parallel: Making the ported LWT library parallel will mean utilising many Java threads to execute the LWT threads. These LWT threads are as mentioned very lightweight and are frequently spawned, Java threads on the other hand are not lightweight and as such a one-to-one mapping of LWT to Java threads will not be possible. To resolve this issue, and thus connect the two sides of the equation, an appropriate scheduler must be written to distribute LWT threads between the fewer Java threads that are active on each core.

Benchmarking: The final aspect of the project is to evaluate how the scalability of software written in LWT scales when run under my ported version as opposed to when run under default OCaml. It will be necessary to convert existing parallel processing benchmarks to use the ported LWT library and also test the scalability on hardware of varying parallel processing power, such as on a dual-core, quad-core and 48-core machine.

8.3 Success Criteria

For the project to be a success I have set the following requirements:

1. I am able to show the scalability difference between two programs using the same piece of LWT code, one compiled using the original LWT libraries under the normal OCaml compiler and one compiled using my ported LWT libraries under OCaml-Java, running on the JVM.

2. Parallel processing benchmark can be used to determine the scalability using LWT under OCaml-Java and using LWT under the normal OCaml environment.

8.4 Starting Point

My experience in functional programming extends as far as the ML courses in Part IA, however I enjoyed completing Project Euler challenges in ML (of which I was normally one of the few people to do so!). Neither have I developed code of a substantial size in ML or OCaml, therefore the best place place to begin would be looking into the structure of open source OCaml code and get a feel for how things are laid out in OCaml. In addition the book 'Real World OCaml' [13] has a public beta available before publication (lucky for me) which will also provide a solid foundation in producing real world OCaml code.

This project relies on the OCaml-Java project, an OCaml to Java bytecode compiler, developed by Xavier Clerc at INRIA. This project provides the ability to use the numerous Java libraries as well as Java's already mentioned parallel processing capabilities. Furthermore the JVM can be found installed on many different types of hardware and systems meaning the target hardware for OCaml code will be greatly increased by using OCaml-Java. Xavier Clerc, having strong connections with the OCaml Labs here in Cambridge, is also keen to support this project. The version of OCaml-Java I will be using is an early preview release of version 2.0.

Finally getting to grips with the innards of LWT will be most easily achievable by experimenting with the software first hand and diving into the source code available on Github.

8.5 Optional Extensions

Some ideas for optional extensions include:

- Experiment with different schedulers to see which perform better and investigate why.
- Attempt to get higher compatibility rate than initially decided which would mean ironing out incompatibilities with current software written with LWT when run with my ported libraries.

8.6 Timetable and Milestones

17th Oct – 6th Nov Further research into LWT's source. Research OCaml coding techniques.
Milestone: Complete LWT coding tutorial; Read through 'Real World OCaml'; Implement some parallel programs in OCaml-Java 2.0; ready to begin planning.

7th Nov – 27th Nov Continued research into LWT's source. Plan implementation of basic functionality of the ported library.
Milestone: Ready to begin development.

28th Nov – 18th Dec Begin porting LWT to single threaded interfacing with OCaml-Java.
Milestone: Some basic functionality implemented.

19th Dec – 8th Jan Continued porting of LWT to single threaded interfacing with OCaml-Java.
Milestone: Porting complete. Should be able to walk through the LWT tutorial without much/any hassle.

9^{th} Jan – 29^{th} Jan Begin creating the scheduler to manage the distribution of LWT threads to OCaml-Java threads.
Milestone: Fixed number of Java threads are able to be scheduled to, regardless of number of cores on system.

30^{th} Jan – 19^{th} Feb Continued scheduler development.
Milestone: Number of Java threads scales with the number of cores present on the system.

20^{th} Feb – 12^{th} Mar Begin conversion of parallel processing benchmarks.
Milestone: Most of benchmarking software conversion complete, fixed relevant bugs which may arise from testing with the benchmark.

13^{th} Mar – 2^{nd} April Testing with benchmarks, finishing dissertation evaluation and write-up.
Milestone: Relevant scalability graph data accumulated.

3^{rd} April – 23^{rd} April Finalising dissertation write-up.
Milestone: Dissertation complete!

8.7 Resources Required and Backup

A list of required resources:

- My own Machines:
  - Macbook Air 13” 2013 (8GB RAM, 512GB Storage, OSX/Windows 8)
  - Desktop Computer (Ubuntu 13.04/Windows 7, Intel Core2 Quad, 5TB Storage)
- Machines in College
- A many-core machine in the SRG (32 - 48 Cores; e.g. ‘Roo’)
- OCaml-Java and Sources[1]
- OCaml[2]
- LWT Sources[3]

[1] OCaml-Java: http://ocamljava.x9c.fr/
[2] OCaml: http://ocaml.org
[3] LWT: http://ocsigen.org/lwt/

Backups will be provided by Dropbox, Skydrive, my own personal USB stick and my personal storage on the PWF facilities. Cloud backup will be incredibly useful for keeping my data accessible from many locations and also avoiding local corruption which could occur at any time. Version control (and also the main copy of my dissertation/project) will be provided by Github. I chose Github since I am already very familiar with git version control; Github provides cloud storage so I'm able to retrieve and work on my files from various locations; furthermore Github provides free private repositories for students.