Tamás Kispéter

Monadic Concurrency in OCaml

Part II in Computer Science

Churchill College

April 16, 2014

Proforma

Name:	**Tamás Kispéter**
College:	**Churchill College**
Project Title:	**Monadic Concurrency in OCaml**
Examination:	**Part II in Computer Science, July 2014**
Word Count:	**1587**[1] **(well less than the 12000 limit)**
Project Originator:	Tamás Kispéter
Supervisor:	Jeremy Yallop

Original Aims of the Project

To write an OCaml framework for lightweight threading. This framework should be defined from basic semantics and have these semantics represented in a theorem prover setting for verification. The verification should include proofs of basic monadic laws. This theorem prover representation should be extracted to OCaml where the extracted code should be as faithful to the representation as possible. The extracted code should be able to run OCaml code concurrently.

Work Completed

All that has been completed appears in this dissertation.

Special Difficulties

Learning how to incorporate encapulated postscript into a LaTeX document on both CUS and Thor.

[1] This word count was computed by `detex diss.tex | tr -cd ’0-9A-Za-z \n’ | wc -w`

Declaration

I, Tamás Kispéter of Churchill 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 April 16, 2014

Contents

List of Figures

Listings

Acknowledgements

Chapter 1

Introduction

The goal of this project is to build a concurrency framework for OCaml. This framework will be designed with correctness in mind: developing the well defined semantics, modelled in a proof assistant and finally extracted to actual code. This project aims to be a verifiable reference implementation.

1.1 Motivation

Verification of core libraries is becoming increasingly important as we discover more and more subtle bugs that even extensive unit testing could not find. As Dijkstra said, testing shows the presence, not the absence of bugs. On the other hand verification can show the absence of bugs, at least with respect to the formal model of the system.

Motivation of the project is to investigate the lack of certified implementation of a concurrency framework. Verified concurrent systems have been researched for languages like C[24], C++ and Java[20], but not yet for OCaml.

1.2 Overview of concurrency

Concurrency is the concept of more than one thread of execution making progress in the same time period. A particular form of concurrency is parallelism, when threads physically run simultaneously.

Concurrent computation has became the norm for most applications in computer schience with the rise of faster systems often with multiple cores. This can be exploited on multiple levels ranging from hardware supported instruction and thread level parallelism to software based heavy and lightweight models.

This project aims to model lightweight, cooperative concurrency. No threads are exposed to the underlying operating system or hardware. This means that blocking operations on the process level will still block all internal threads. The threads expose the points of possible interleaving and the scheduling is done in software.

The issue of concurrency occurs in most general programming languages. However, functional languages are often both less developed in this area and also fit well with concurrency without races for mutable data structures. Functional languages that have both actual industrial applications and large set of features are of particular interest. These languages include OCaml and Haskell. This project focuses on OCaml.

1.3 Overview of OCaml

OCaml is a high level programming language. It combines functional, object-oriented and imperative paradigms and used in large scale industrial and academic projects where speed and correctness are of utmost importance. OCaml uses one of the most powerful type and inference systems available to make efficient and correct software engineering possible.

1.4 Current implementations of a concurrency framework in OCaml

There are two very successful monadic concurrency frameworks, LWT[2] and Async[27]. They both provide the primitives for concurrent development in OCaml, however neither is supported by a clear semantic description. This is because their main focus was ease of use and speed.

LWT, the lightweight cooperative threading library[29] was designed as an open source framework entirely written in OCaml in a monadic style. It was successfully used in several large projects including the Unison file synchroniser and the Ocsigen Web server. This library includes many primitives to provide a feature rich framework, including primitives for thread creation, composition and cancel, thread local storage and support for various synchronisation techniques.

```ocaml
open Lwt

let main () =
  let heads =
    Lwt_unix.sleep 1.0 >>
    return (print_endline "Heads");
  in
  let tails =
    Lwt_unix.sleep 2.0 >>
    return (print_endline "Tails");
  in
  lwt () = heads <&> tails in
  return (print_endline "Finished")

let _ = Lwt_main.run (main ())
```

Listing 1.1: LWT example

In this example we can see some of the syntax of Lwt where we define `heads`, a function that sleeps for 1 second and then prints “Head” and `tails` which does the same with 2 seconds and “Tails”. These are then both waited on, until they are both finished and followed by the text “Finished”. In Lwt semantics mostly follow the principle of a continuation monads, that is we build a sequence of continuations that the scheduler can pick between.

An other implementation, Async is an open source concurrency library for OCaml developed by Jane Street. Unlike LWT the basic semantics are designed with the promise monads in mind.

```ocaml
open Jane.Std
open Async.Std

let heads  =  (after (sec 1.0)  >>| fun () -> (print_endline "Heads"));;
let tails = (after (sec 2.0)  >>| fun () -> (print_endline "Tails"));;
let head_and_tails = (Deferred.both
   heads
   tails);;


let () = upon (head_and_tails) (fun _ -> ());;

let () = never_returns (Scheduler.go ());;
```

Listing 1.2: Async example

In this snippet we define `heads` and `tails` as Deferred values of the respective code sequences. A Deferred value is the promise of that value evaluating. It can be used in place of a value of the same type and binding on it returns a deferred value that is when is fulfilled applies the bound function.

There are a number of other experimental implementations for concurrency in OCaml, including JoCaml[21] that implements join calculus over Ocaml, Functory[14] that focuses on distributed computation, Ocamlnet that exploits multiple cores and OCamlMPI[16] which provides bindings for the standard MPI message passing framework.

1.5 Semantics of concurrency

There have been a lot of work on how exactly to formulate the semantics of concurrent and distributed systems. Some of the most common models for lightweight concurrency[10] are captured[11] and delimited[15] continuations[25], trampolined style[12], continuation monads[9], promise monads[19] and event based programming(used in the OCamlNet[26] project). This work focuses on the continuation monad style.

A monad[13] in functional programming is a construct to structure computations that are in some sense "sequenced" together. This sequencing can be for example string concatenation, simple operation sequencing (the well known ;) or conditional execution. Two operations, commonly called bind and return form a monad when they obey a set of axioms called monadic laws.

The continuation monad consists of a parametric type $\alpha\,\mathbf{con}$, where α is a type parameter describing the type of computation or value enclosed and three key operations

- ret, also known as return. It has type $\alpha \rightarrow \alpha\,\mathbf{con}$ and simply evaluates its parameter and boxes up the result in the parametric type

- >>=, also known as bind, sequences two operations. The second argument is the continuation for the first parameter. More formally it has a type $\alpha\,\mathbf{con} \rightarrow (\alpha \rightarrow \beta\,\mathbf{con}) \rightarrow \beta\,\mathbf{con}$, that is it takes a boxed up computation and a function that takes the value of the computation and returns a new box. Bind then evaluates the expression within the box of the first argument and passes it to the second argument.

The third operation is fork, which is the way to spawn new threads (two in this particular case). The approach taken in this work is to have fork take two arguments and evaluate the two paths concurrently. The concurrency is achieved by reducing either side of the fork step by step based on some scheduler. When either path reduced to a value it returns a boxed up pair of results. This result is value and a boxed up computation (that is, the partially reduced other path).

Therefore the signature of fork is slightly more complicated

$$\mathbf{fork} \,:\, \alpha\,\mathbf{con} \rightarrow \beta\,\mathbf{con} \rightarrow ((\alpha * \beta\,\mathbf{con}) \,+\, (\alpha\,\mathbf{con} * \beta))\,\mathbf{con}$$

Where + refers to sum types, that is labelled variants.

1.6 Semantics to logic

These semantics can be modelled in logic, in particular logics used by proof assistants. A model like this can be used for formal verification of properties about said semantics [6, 8, 7, 17]. Most often used proof assistants include Coq [4], HOL and Isabelle. There has been much effort to make the modelling as easy as possible with tools like Ott[22] which enables semanticists to describe their language in a simple, ascii-art like style and export these models to destination languages, including the above mentioned proof assistants and LaTeX.

1.7 Logic to runnable code

While a number of proof assistants have utilities for direct computation, in most cases semantics is described as a set of logical relations. This representation is more amenable to proofs than to actual execution. Much research has gone into extraction of these representations as certified programs in functional programming languages like OCaml [18] and Haskell. The extracted code then can be run and the certification is a proof that the code is faithful to the representation in the proof assistant.

Chapter 2

Preparation

During the preparation phase of this project many decisions had to be made, including the concurrency model, large scale semantics and the tool chain used in the process.

2.1 Design of monadic semantics

Monads: Brief mathematical background.

Category theory has been a general tool used to model functional programming languages and programs. Monads are a concept originating from this connection.

A monad on a category C is a triple (T, η, μ) where T is an endofunctor on C, that is, it maps the category to itself. The last two, η and μ are natural transformations such that $\eta : 1_C \to T$, that is between the identity functor and T, and $\mu : T^2 \to T$. The first transformation, η describes a lift operation: essentially we can wrap the object in C and preserving its properties. The second transformation, μ is about an operation called join. This operation unwraps a layer of wrapping if there are two. To call a triple like this a monad it has to satisfy two conditions, called coherence conditions.

1.

$$\mu \circ T\mu = \mu \circ \mu T$$

Or as commutative diagram:

$$\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

This roughly demands that unwrapping from three layers to one is associative.

2. $$\mu \circ T\eta = \mu \circ \eta T = 1_T$$

   Or as commutative diagram:

$$\begin{tikzcd}
T \arrow[r, "\eta T"] \arrow[d, "T\eta"'] \arrow[dr, equal] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

   This roughly translates to wrapping and then subsequently unwrapping behaves as an identity.

This description entails two operations (lift and join) and their behaviour (associativity and identity), however this is a formulation rarely used in practice. There is an equivalent pair of operations (ret and bind) with similar behaviour constraints that is used in most monadic programming constructs.

The ret takes a value of the language and gives its monadic counterpart. With types this can be represented as $\mathbf{ret} : \forall\alpha.\alpha \to \mathbf{con}\,\alpha$.

The bind

To call this system a monad, we need to satisfy three axioms:

1. ret is essentially a left neutral element:

$$(\mathbf{ret}\, x) \gg= f \quad \equiv \quad f\, x$$

2. ret is essentially a right neutral element:

$$m \gg= \mathbf{ret} \quad \equiv \quad m$$

3. bind (often written as $\gg=$) is associative:

$$(m \gg= f) \gg= g \quad \equiv \quad m \gg= (\lambda\, x.(f\, x \gg= g))$$

We will return to the exact nature of $\equiv$ used in this project in the evaluation section and the exact form this will take.

2.2 Choice of concurrency model

There are a number of lightweight concurrency models to choose from.

2.3 Design of concurrent semantics

Concurrency in a language can be modelled in various ways

2.4 Tools

The project uses a chain of three tools:

1. Ott, a tool for transforming informal, readable semantics to both LaTeXand formal proof assistant code.
2. Coq, a proof assistant supported by Ott.
3. OCaml, the target language.

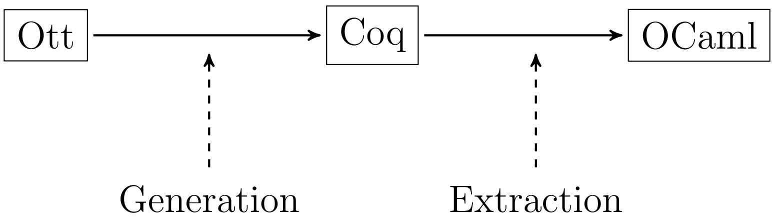

Figure 2.1: Toolchain outline

In the preparation phase I got acquainted with all three of these systems, as I have not used them before for any serious work.

2.4.1 Ott

To avoid duplication of the semantics in several formats I have to chosen to use a supporting tool called Ott. It enables the use of a simple ASCII-art like description of grammars, typing and reduction relations. Ott can export to various destination formats including most proof assistants and LaTeX. This is the primary form of the semantics that all further forms are derived from in the project.

For someone familiar to formal semantics Ott has an easy to use and intuitive syntax.

Metavariables used in productions are defined with their destination language equivalents and potentially (in the case of Coq) their equality operation

```
metavar termvar, x ::=   {{ com  term variable }}
{{ isa string}} {{ coq nat}} {{ hol string}} {{ coq-equality }}
{{ ocaml int}} {{ lex alphanum}} {{ tex \mathit{[[termvar]]} }}
```

Listing 2.1: Ott metavariable definition

Term expression grammars and other grammars can be defined in the well known Backus-Naur form with some extensions.

```
grammar
t :: 't_' ::=                                      {{ com term     }}
  | x              ::   :: Var                        {{ com variable}}
  | \ x . t        ::   :: Lam (+ bind x in t +) {{ com lambda   }}
  | t t'           ::   :: App                        {{ com app      }}
  | ( t )          :: S:: Paren                      {{ icho [[t]]   }}
  | { t / x } t'   :: M:: Tsub
                         {{ icho (tsubst_t [[t]] [[x]] [[t']])}}
```

Listing 2.2: Ott grammar example

In this example the non-terminal t for terms is defined with 5 productions: variables, lambda abstractions, applications, parentheses grouping and variable substitution. Each of these rules have a name, for example Var and Lam. Each of these names are prefixed by the unique prefix t_ to have non-ambiguous names. The right hand side of each line describes the translation to target languages, for example com will generate the given description for the LaTeX target.

There are meta flags S and M to describe syntactical sugar and meta productions that are not generated as data structure elements in target languages, but instead have their own instructions: for example the substitution term will be rewritten as an application of the tsubst_t relation defined elsewhere.

In many languages one might want to define a value subgrammar, which can be used both in the reduction relation definition and in proving properties of the semantics. Ott has support for general subgrammar relation check.

```
v :: 'v_' ::=                                      {{ com value    }}
  | \ x . t        ::   :: Lam                        {{ com lambda   }}

subrules
  v <:: t
```

Listing 2.3: Ott value subgrammar example

Here v is a subgrammar of t. The statement v <:: t is exported as a target language subroutine that checks whether this holds and during translation it checks for obvious bugs.

Another common feature of semantics is substitution of values for variables, for example in function application. This is so frequent that Ott provides both single and multiple variable substitutions for the target languages as subroutines in the translated code.

```
substitutions
  single t x :: tsubst
```

Listing 2.4: Ott substitution example

The statement single t x :: tsubsts defines a single substitution function called tsubsts_t over terms defined by the grammar for t and for variables represented by the metavariable x. This is the relation mentioned in the grammar for the target language version for { t / x } t'.

Finally paramount to most semantics are relations like the reduction relation.

```
defns
Jop :: '' ::=

 defn
 t1 --> t2 :: ::reduce::'' {{ com [[t1]] reduces to [[t2]]}} by


    ------------------------------   :: ax_app
    (\x.t12) v2 -->  {v2/x}t12

    t1 --> t1'
    ----------------  :: ctx_app_fun
    t1 t --> t1' t

    t1 --> t1'
    ----------------  :: ctx_app_arg
    v t1 --> v t1'
```

Listing 2.5: Ott reduction relation example

In this example I define a set of mutually recursive relations named Jop with one relation in it the −− > or reduce relation. Each element of this relation takes the form t1 −− > t2, where t1 and t2 are both terms of the grammar defined above. There are three statements for function application: the actual substitution, reduction of the first term and reduction of the second term.

```
t1 --> t1'
----------------  :: ctx_app_fun
t1 t --> t1' t
```

Listing 2.6: Ott single reduction

The premise(s) appear line-by-line above the ascii-art line, while and the result below the line. Next to the line is the name of the statement which is then prefixed by the name of the relation to avoid ambiguity.

2.4.2 Coq

There are a number of proof assistants available as destination for Ott, out of which Coq and Isabelle provide good extraction facilities to OCaml. They are at a glance rather similar. The choice between the two came down to advice from supervisors as I did not have experience with either systems. This project was developed with the Coq proof assistant.

Coq is formal proof assistant with a mathematical higher-level language called *Gallina*, based around the Calculus of Inductive Constructions, that can be used to define functions and predicates, state, formally prove and machine check mathematical theorems and extract certified programs to high level languages like Haskell and OCaml.

Objects in Coq can divided into two sorts, *Prop* (propositions) and *Type*. A proposition like $\forall A, B.\ A \wedge B \to B \vee B$ translates as

```
forall A B : Prop, A ∧ B → B ∨ B
```

Listing 2.7: Coq Prop logic example

Predicates like equality and other sets can be used as well

```
forall x y : Z, x * y = 0 → x = 0 ∨ y = 0
```

Listing 2.8: Coq Prop predicate example

New predicates can be defined inductively

```
Inductive even : N → Prop :=
 | even_0 : even 0
 | even_S n : odd n → even (n + 1)
 with odd : N → Prop :=
 | odd_S n : even n → odd (n + 1).
```

Listing 2.9: Coq Prop new predicate example

Data structures can also be defined both inductively and coinductively

```
Inductive seq : nat → Set :=
 | niln : seq 0
 | consn : forall n : nat, nat → seq n → seq (S n).
```

Listing 2.10: Coq inductive data structure example

```
CoInductive stream (A:Type) : Type :=
 | Cons : A → stream → stream.
```

Listing 2.11: Coq coinductive data structure example

Functions over these data structures are defined as fixpoints and cofixpoints respectively.

```
Fixpoint length (n : nat) (s : seq n) {struct s} : nat :=
    match s with
    | niln ⇒ 0
    | consn i _ s' ⇒ S (length i s')
    end.
```

Listing 2.12: Coq fixpoint example

Finally theorems can be proven with these propositions and structures.

```
Theorem length_corr : forall (n : nat) (s : seq n), length n s = n.
  Proof.
    intros n s.

    (* reasoning by induction over s. Then, we have two new goals
       corresponding on the case analysis about s (either it is
       niln or some consn *)
    induction s.

      (* We are in the case where s is void. We can reduce the
         term: length 0 niln *)
      simpl.

      (* We obtain the goal 0 = 0. *)
      trivial.

      (* now, we treat the case s = consn n e s with induction
         hypothesis IHs *)
      simpl.

      (* The induction hypothesis has type length n s = n.
         So we can use it to perform some rewriting in the goal: *)
      rewrite IHs.

      (* Now the goal is the trivial equality: S n = S n *)
      trivial.

    (* Now all sub cases are closed, we perform the ultimate
       step: typing the term built using tactics and save it as
       a witness of the theorem. *)
  Qed.
```

Listing 2.13: Coq theorem example

Each Lemma, Theorem, Example have a name and a statement. The statement is a proposition. This is followed by the proof in which a sequence of steps modify the assumed hypotheses and the goal proposition until it has been proven. These steps are called tactics which can be simple application of previous theorems and axioms or as complex as a SAT solver. Coq comes with a language Ltac to allow users to build their own tactics.

Coq also provides built in facilities for the certified extraction of code to OCaml, Haskell and Scheme. These can be invoked with the keywords Extraction and Recursive Extraction.

```
type nat =
| O
| S of nat

type seq =
| Niln
| Consn of nat * nat * seq
```

Listing 2.14: Coq to OCaml extraction of seq

```
(** val length : nat -> seq -> nat **)

let rec length n = function
| Niln -> O
| Consn (i, n0, s') -> S (length i s')
```

Listing 2.15: Coq to OCaml extraction of length

Out of the box, Coq does not provide facilities for the extraction of so called logical inductive systems. These are essentially inductively defined propositions.

```
Inductive add : nat → nat → nat → Prop :=
| addO : forall n , add n O n
| addS : forall n m p, add n m p → add n (S m) (S p).
```

Listing 2.16: Coq logical inductive example

However with the help of a plugin developed by David Delahaye, Catherine Dubois, Jean-Frédéric Étienne and Pierre-Nicolas Tollitte [28] by marking different modalities of the elements of a relation we can generate code with an input-output convention.

2.4.3 OCaml

Chapter 3

Implementation

3.1 The semantics

3.1.1 Expressions

Arrow types

Sum types

Product types

Fixpoint combinator

Monadic primitives

Fork

Computation placeholders

3.1.2 Transition system

Labelled transitions

Select operator

3.1.3 Type system

3.2 Proof assisstant system

3.2.1 Outline of the proof assistant code

3.2.2 Modification

3.2.3 Extractable version

3.3 OCaml system

3.3.1 Outline of the OCaml code

Chapter 4

Evaluation

4.1 Theoretical evaluation

Properties to evaluate: monadic laws, fork commutativity and associativity (liveness ?), type preservation and progress ?.

4.1.1 Methods

Weak bisimilarity

Intro to weak bisimilarity.

What form does a general weak bisimilarity proof take.

How does it appear here.

4.1.2 Properties

Monadic laws

Fork commutativity

Outline of fork commutativity

Fork associativity

Liveness?

Type preservation?

Progress?

4.2 Practical evaluation

Evaluation of speed and memory requirements absolutely and relative to implementations in LWT and Async.

4.2.1 Methods

4.2.2 Examples

Kahn process network

Eratosthene Sieve

Concurrent sort

Chapter 5

Conclusion

I hope that this rough guide to writing a dissertation is LaTeX has been helpful and saved you time.

Bibliography

[1] Lem, a tool for lightweight executable mathematics. `http://www.cs.kent.ac.uk/people/staff/sao/lem/`.

[2] Lwt, lightweight threading library. `http://ocsigen.org/lwt/`.

[3] Ocaml. `http://ocaml.org/`.

[4] Ott, a tool for writing definitions of programming languages and calculi. `http://coq.inria.fr/`.

[5] Ott, a tool for writing definitions of programming languages and calculi. `http://www.cl.cam.ac.uk/~so294/ocaml/`, 2008.

[6] Nick Benton and Vasileios Koutavas. A mechanized bisimulation for the nu-calculus. *Higher-Order and Symbolic Computation (to appear, 2013)*, 2008.

[7] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a c compiler front-end. In *FM 2006: Formal Methods*, pages 460–475. Springer, 2006.

[8] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[9] Koen Claessen. Functional pearls: A poor man's concurrency monad, 1999.

[10] Christophe Deleuze. Light weight concurrency in ocaml: Continuations, monads, promises, events.

[11] Daniel P Friedman. Applications of continuations. In *Proceedings of the ACM Conference on Principles of Programming Languages*, 1988.

[12] Steven E Ganz, Daniel P Friedman, and Mitchell Wand. Trampolined style. In *ACM SIGPLAN Notices*, volume 34, pages 18–27. ACM, 1999.

[13] CAR Hoareetal. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. *Engineering theories of software construction*, 180:47, 2001.

[14] Jean-Christophe Filliâtre K Kalyanasundaram. Functory. `https://www.lri.fr/~filliatr/functory/About.html`, 2010.

[15] Oleg Kiselyov. Delimited control in ocaml, abstractly and concretely: System description. In *Functional and Logic Programming*, pages 304–320. Springer, 2010.

[16] Xavier Leroy. Ocamlmpi: Interface with the mpi message-passing interface.

[17] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[18] Pierre Letouzey. Extraction in coq: An overview. In *Logic and Theory of Algorithms*, pages 359–369. Springer, 2008.

[19] Barbara Liskov and Liuba Shrira. *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*, volume 23. ACM, 1988.

[20] Andreas Lochbihler. *A Machine-Checked, Type-Safe Model of Java Concurrency: Language, Virtual Machine, Memory Model, and Verified Compiler.* KIT Scientific Publishing, 2012.

[21] Louis Mandel and Luc Maranget. The JoCaml system. `http://jocaml.inria.fr/`, 2007.

[22] Francesco Zappa Nardelli. Ott, a tool for writing definitions of programming languages and calculi. `http://www.cl.cam.ac.uk/~pes20/ott/`.

[23] Scott Owens. A sound semantics for ocaml light. In *Programming Languages and Systems*, pages 1–15. Springer, 2008.

[24] Jaroslav Ŝevčik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In *ACM SIGPLAN Notices*, volume 46, pages 43–54. ACM, 2011.

[25] Chung-chieh Shan. Shift to control. In *Proceedings of the 5th workshop on Scheme and Functional Programming*, pages 99–107, 2004.

[26] Gerd Stolpmann. Ocamlnet. `http://projects.camlcity.org/projects/ocamlnet.html`.

[27] Jane Street. Async, open source concurrency library.

[28] Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. Producing certified functional code from inductive specifications. In *Certified Programs and Proofs*, pages 76–91. Springer, 2012.

[29] Jérôme Vouillon. Lwt: a cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 3–12. ACM, 2008.

Appendix A

Ott source

A.1 diss.tex

```
% The master copy of this demo dissertation is held on my filespace
% on the cl file serve (/homes/mr/teaching/demodissert/)

% Last updated by MR on 2 August 2001

\documentclass[12pt,twoside,notitlepage]{report}

\usepackage{a4}
\usepackage{verbatim}
\usepackage{etoolbox}

\newtoggle{colourtoggle}

\toggletrue{colourtoggle}

\usepackage{url}
\input{epsf}                            % to allow postscript inclusions
% On thor and CUS read top of file:
%     /opt/TeX/lib/texmf/tex/dvips/epsf.sty
% On CL machines read:
%     /usr/lib/tex/macros/dvips/epsf.tex



\raggedbottom                           % try to avoid widows and orphans
\sloppy
\clubpenalty1000%
\widowpenalty1000%

\addtolength{\oddsidemargin}{6mm}       % adjust margins
\addtolength{\evensidemargin}{-8mm}

\renewcommand{\baselinestretch}{1.1}    % adjust line spacing to make
                                        % more readable
%  Listings set up
\usepackage{listings, lstlangcoq, lstlangott, bold-extra}
\usepackage[usenames,dvipsnames,svgnames,table]{xcolor}

\iftoggle{colourtoggle}{
\definecolor{mygreen}{rgb}{0,0.6,0}
\definecolor{myblue}{rgb}{0,0,1}
\definecolor{myemerald}{rgb}{0.0,0.66,0.42}
\definecolor{mygray}{rgb}{0.5,0.5,0.5}
\definecolor{mymauve}{rgb}{0.58,0,0.82}
\definecolor{myseagreen}{rgb}{0.95,0.999,0.95}
}{
\definecolor{mygreen}{rgb}{0.35,0.35,0.35}
\definecolor{myblue}{RGB}{5,5,5}
\definecolor{mygray}{rgb}{0.5,0.5,0.5}
\definecolor{myemerald}{RGB}{111,111,111}
\definecolor{mymauve}{rgb}{0.25,0.25,0.25}
\definecolor{myseagreen}{rgb}{0.98,0.98,0.98}
}

\lstset{ %
  backgroundcolor=\color{myseagreen},   % choose the background color; you must add \usepackage{color} or \usepackage{xcolor}
  basicstyle=\footnotesize,        % the size of the fonts that are used for the code
  breakatwhitespace=false,         % sets if automatic breaks should only happen at whitespace
  breaklines=true,                 % sets automatic line breaking
  captionpos=b,                    % sets the caption-position to bottom
  commentstyle=\color{mygreen},    % comment style
  deletekeywords={...},            % if you want to delete keywords from the given language
  escapeinside={\%*}{*)},          % if you want to add LaTeX within your code
  extendedchars=true,              % lets you use non-ASCII characters; for 8-bits encodings only, does not work with UTF-8
  frame=shadowbox,                    % adds a frame around the code
  frameround=tttt,
  keepspaces=true,                 % keeps spaces in text, useful for keeping indentation of code (possibly needs columns=flex
  keywordstyle=\color{myblue},       % keyword style
  language=[Objective]Caml,        % the language of the code
  morekeywords={*,...},            % if you want to add more keywords to the set
  numbers=left,                    % where to put the line-numbers; possible values are (none, left, right)
  numbersep=5pt,                   % how far the line-numbers are from the code
  numberstyle=\tiny\color{myemerald}, % the style that is used for the line-numbers
  rulecolor=\color{myemerald},         % if not set, the frame-color may be changed on line-breaks within not-black text (e.g.
  showspaces=false,                % show spaces everywhere adding particular underscores; it overrides 'showstringspaces'
  showstringspaces=false,          % underline spaces within strings only
  showtabs=false,                  % show tabs within strings adding particular underscores
  stepnumber=1,                    % the step between two line-numbers. If it's 1, each line will be numbered
  stringstyle=\color{mymauve},     % string literal style
  tabsize=2,                       % sets default tabsize to 2 spaces
  title=\lstname                   % show the filename of files included with \lstinputlisting; also try caption instead of ti
}

% Tikz set up
\usepackage{tikz}
\usepackage{tikz-cd}
\usetikzlibrary{arrows, shapes}

\begin{document}

\bibliographystyle{plain}
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title


\pagestyle{empty}

\hfill{\LARGE \bf Tam\'as Kisp\'eter}

\vspace*{60mm}
\begin{center}
\Huge
{\bf Monadic Concurrency in OCaml} \\
\vspace*{5mm}
Part II in Computer Science \\
\vspace*{5mm}
Churchill College \\
\vspace*{5mm}
\today  % today's date
\end{center}

\cleardoublepage

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Proforma, table of contents and list of figures

\setcounter{page}{1}
\pagenumbering{roman}
\pagestyle{plain}

\chapter*{Proforma}

{\large
\begin{tabular}{ll}
Name:               & \bf Tam\'as Kisp\'eter                     \\
College:            & \bf Churchill College                     \\
Project Title:      & \bf Monadic Concurrency in OCaml \\
Examination:        & \bf Part II in Computer Science, July 2014        \\
Word Count:         & \bf 1587\footnotemark[1]
(well less than the 12000 limit) \\
Project Originator: & Tam\'as Kisp\'eter                    \\
Supervisor:         & Jeremy Yallop                    \\
\end{tabular}
}
\footnotetext[1]{This word count was computed
by {\tt detex diss.tex | tr -cd '0-9A-Za-z $\tt\backslash$n' | wc -w}
}
\stepcounter{footnote}


\section*{Original Aims of the Project}

To write an OCaml framework for lightweight threading. This framework should be defined from basic semantics and have th


\section*{Work Completed}

All that has been completed appears in this dissertation.

\section*{Special Difficulties}
```

```
Learning how to incorporate encapulated postscript into a \LaTeX\
document on both CUS and Thor.

\newpage
\section*{Declaration}

I, Tam\'as Kisp\'eter of Churchill 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.

\bigskip
\leftline{Signed [signature]}

\medskip
\leftline{Date \today}

\cleardoublepage

\tableofcontents

\listoffigures

\lstlistoflistings

\newpage
\section*{Acknowledgements}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% now for the chapters

\cleardoublepage        % just to make sure before the page numbering
                        % is changed

\setcounter{page}{1}
\pagenumbering{arabic}
\pagestyle{headings}
% Proforma + Intro + prep = 3100
% Implementation = 4700
% Evaluation + conclustion = 2200
% Appendicies = \infty
\chapter{Introduction}

The goal of this project is to build a concurrency framework for OCaml. This framework will be designed with correctness in mi

\section{Motivation}
Verification of core libraries is becoming increasingly important as we discover more and more subtle bugs that even extensive

Motivation of the project is to investigate the lack of certified implementation of a concurrency framework. Verified concurre

\section{Overview of concurrency}
Concurrency is the concept of more than one thread of execution making progress in the same time period. A particular form of

Concurrent computation has became the norm for most applications in computer schience with the rise of faster systems often wi

This project aims to model lightweight, cooperative concurrency. No threads are exposed to the underlying operating system or
```

```
The issue of concurrency occurs in most general programming languages. However, functional languages are often both less

\section{Overview of OCaml}
OCaml is a high level programming language. It combines functional, object-oriented and imperative paradigms and used in


\section{Current implementations of a concurrency framework in OCaml}
There are two very successful monadic concurrency frameworks, LWT\cite{LWT} and Async\cite{Async}. They both provide the

LWT, the lightweight cooperative threading library\cite{vouillon2008lwt} was designed as an open source framework entirel

\begin{minipage}{\linewidth}
\begin{lstlisting}[caption={LWT example}]
 open Lwt

 let main () =
   let heads =
     Lwt_unix.sleep 1.0 >>
     return (print_endline "Heads");
   in
   let tails =
     Lwt_unix.sleep 2.0 >>
     return (print_endline "Tails");
   in
   lwt () = heads <&> tails in
   return (print_endline "Finished")

 let _ = Lwt_main.run (main ())
\end{lstlisting}
\end{minipage}

In this example we can see some of the syntax of Lwt where we define \verb|heads|, a function that sleeps for 1 second a

An other implementation, Async is an open source concurrency library for OCaml developed by Jane Street. Unlike LWT the

\begin{lstlisting}[caption={Async example}]
open Jane.Std
open Async.Std

let heads  =  (after (sec 1.0)  >>| fun () -> (print_endline "Heads"));;
let tails = (after (sec 2.0)  >>| fun () -> (print_endline "Tails"));;
let head_and_tails = (Deferred.both
   heads
   tails);;


let () = upon (head_and_tails) (fun _ -> ());;

let () = never_returns (Scheduler.go ());;
\end{lstlisting}

In this snippet we define \verb|heads| and \verb|tails| as Deferred values of the respective code sequences. A Deferred


There are a number of other experimental implementations for concurrency in OCaml, including JoCaml\cite{jocaml} that im


\section{Semantics of concurrency}
```

```
There have been a lot of work on how exactly to formulate the semantics of concurrent and distributed systems. Some of the most

A monad\cite{hoareetal2001tackling} in functional programming is a construct to structure computations that are in some sense


The continuation monad consists of a parametric type $ \alpha \, \textbf{con} $, where $ \alpha $ is a type parameter describi
\begin{itemize}
\item{ret, also known as return. It has type $ \alpha \rightarrow \alpha \, \textbf{con} $ and simply evaluates its parameter
\item{$>>=$, also known as bind, sequences two operations. The second argument is the continuation for the first parameter. Mo

\end{itemize}

The third operation is fork, which is the way to spawn new threads (two in this particular case). The approach taken in this w

Therefore the signature of fork is slightly more complicated
\[ \textbf{fork} \, : \, \alpha \, \textbf{con} \rightarrow \beta \, \textbf{con} \rightarrow ((\alpha *  \beta \, \textbf{con
Where $ + $ refers to sum types, that is labelled variants.

\section{Semantics to logic}
These semantics can be modelled in logic, in particular logics used by proof assistants. A model like this can be used for for


\section{Logic to runnable code}
While a number of proof assistants have utilities for direct computation, in most cases semantics is described as a set of log



\chapter{Preparation}
During the preparation phase of this project many decisions had to be made, including the concurrency model, large scale seman

\section{Design of monadic semantics}
%Brief overview of monads
Monads: Brief mathematical background.

Category theory has been a general tool used to model functional programming languages and programs. Monads are a concept orig


A monad on a category $ C $ is a triple $ (T, \eta, \mu) $ where $ T $ is an endofunctor on $ C $, that is, it maps the catego
\begin{enumerate}

\item{
\[ \mu \circ T \mu = \mu \circ \mu T \]
Or as commutative diagram:
\begin{center}
\begin{tikzcd}
T^3 \arrow{r}{T\mu} \arrow{d}[swap]{\mu T}
& T^2 \arrow{d}{\mu} \\
T^2 \ar{r}[swap]{\mu} & T
\end{tikzcd}
\end{center}
This roughly demands that unwrapping from three layers to one is associative.



}
\item{
\[ \mu \circ T\eta = \mu \circ \eta T = 1_T \]
Or as commutative diagram:
\begin{center}
\begin{tikzcd}
T \arrow{r}{\eta T} \arrow{d}[swap]{T \eta } \arrow[equal]{rd}
& T^2 \arrow{d}{\mu} \\
T^2 \ar{r}[swap]{\mu} & T
\end{tikzcd}
\end{center}

This roughly translates to wrapping and then subsequently unwrapping behaves as an identity.
}
\end{enumerate}

This description entails two operations (lift and join) and their behaviour (associativity and identity), however this i


The \lstinline|ret| takes a value of the language and gives its monadic counterpart. With types this can be represented 

The \lstinline|bind|

To call this system a monad, we need to satisfy three axioms:

\begin{enumerate}
\item{\lstinline|ret| is essentially a left neutral element:
\[ (\textbf{ret} \, x) \, \gg=\, f \quad \equiv \quad f \, x \]}
\item{\lstinline|ret| is essentially a right neutral element:
\[ m \, \gg=\, \textbf{ret} \quad \equiv \quad m \]}
\item{\lstinline|bind| (often written as $ \gg= $) is associative:
\[ (m \, \gg= \, f) \, \gg= g \quad \equiv \quad m\, \gg= \, (\lambda\, x. (f\, x \, \gg= \, g) ) \]}
\end{enumerate}

We will return to the exact nature of $ \equiv $ used in this project in the evaluation section and the exact form this 



\section{Choice of concurrency model}
There are a number of lightweight concurrency models to choose from.
%Why I chose continuation monads vs. World





\section{Design of concurrent semantics}
%Brief overview of techiniques for concurrency
Concurrency in a language can be modelled in various ways

%Choices for concurrency semantics




\section{Tools}
The project uses a chain of three tools: \begin{enumerate}
\item{
Ott, a tool for transforming informal, readable semantics to both \LaTeX and formal proof assistant code.}
\item{Coq, a proof assistant supported by Ott. }
\item{OCaml, the target language. }
\end{enumerate}

\begin{figure}[h!]
```

```
\centering
\begin{tikzpicture}[node distance=1.8cm, auto, >=stealth',pil/.style={
           ->,
           thick,
           shorten <=2pt,
           shorten >=2pt,}, pildashed/.style={
                      ->,
                      thick,
                      dashed,
                      shorten <=2pt,
                      shorten >=2pt,},scale=1.2, every node/.style={scale=1.2}, ]
\node[draw, rectangle] (o) {Ott};
\node[right of=o] (otch) {};
\node[below of={otch}] (otc) {Generation}
  edge[pildashed] (otch);
\node[draw, rectangle, right of={otch}] (c) {Coq}
  edge[pil, <-] (o);
\node[right of=c] (ceoh) {};
\node[below of={ceoh}] (ceo) {Extraction}
  edge[pildashed] (ceoh);
\node[draw, rectangle, right of={ceoh}] (oc) {OCaml}
  edge[pil, <-] (c);

\end{tikzpicture}
\caption{Toolchain outline}
\end{figure}

In the preparation phase I got acquainted with all three of these systems, as I have not used them before for any serious work
\subsection{Ott}
% Why Ott (because it is awesome)
To avoid duplication of the semantics in several formats I have to chosen to use a supporting tool called Ott. It enables the

For someone familiar to formal semantics Ott has an easy to use and intuitive syntax.

Metavariables used in productions are defined with their destination language equivalents and potentially (in the case of Coq)
\begin{lstlisting}[language={Ott}, caption={Ott metavariable definition}]
metavar termvar, x ::=   {{ com  term variable }}
{{ isa string}} {{ coq nat}} {{ hol string}} {{ coq-equality }}
{{ ocaml int}} {{ lex alphanum}} {{ tex \mathit{[[termvar]]} }}
\end{lstlisting}

Term expression grammars and other grammars can be defined in the well known Backus-Naur form with some extensions.

\begin{lstlisting}[language={Ott}, caption={Ott grammar example}]
grammar
t :: 't_' ::=                               {{ com term    }}
  | x            ::  :: Var                   {{ com variable}}
  | \ x . t      ::  :: Lam (+ bind x in t +) {{ com lambda  }}
  | t t'         ::  :: App                   {{ com app     }}
  | ( t )        :: S:: Paren                 {{ icho [[t]]  }}
  | { t / x } t' :: M:: Tsub
                        {{ icho (tsubst_t [[t]] [[x]] [[t']])}}
\end{lstlisting}

In this example the non-terminal t for terms is defined with 5 productions: variables, lambda abstractions, applications, pare

There are meta flags S and M to describe syntactical sugar and meta productions that are not generated as data structure eleme

In many languages one might want to define a value subgrammar, which can be used both in the reduction relation definition and
```

```
\begin{lstlisting}[language={Ott}, caption={Ott value subgrammar example}]
v :: 'v_' ::=                               {{ com value   }}
  | \ x . t      ::  :: Lam                   {{ com lambda  }}

subrules
  v <:: t
\end{lstlisting}
Here v is a subgrammar of t. The statement v $<::$ t is exported as a target language subroutine that checks whether thi

Another common feature of semantics is substitution of values for variables, for example in function application. This i
\begin{lstlisting}[language={Ott}, caption={Ott substitution example}]
substitutions
  single t x :: tsubst
\end{lstlisting}
The statement \lstinline[language={Ott}]|single t x :: tsubsts| defines a single substitution function called tsubsts\_t



Finally paramount to most semantics are relations like the reduction relation.
\begin{lstlisting}[language={Ott}, caption={Ott reduction relation example}]
defns
Jop :: '' ::=

 defn
 t1 --> t2 :: ::reduce::'' {{ com [[t1]] reduces to [[t2]]}} by


    --------------------------  :: ax_app
    (\x.t12) v2 -->  {v2/x}t12

    t1 --> t1'
    -------------- :: ctx_app_fun
    t1 t --> t1' t

    t1 --> t1'
    -------------- :: ctx_app_arg
    v t1 --> v t1'
\end{lstlisting}
In this example I define a set of mutually recursive relations named Jop with one relation in it the $-->$ or reduce rel
\begin{lstlisting}[language={Ott}, caption={Ott single reduction}]
t1 --> t1'
-------------- :: ctx_app_fun
t1 t --> t1' t
\end{lstlisting}
The premise(s) appear line-by-line above the ascii-art line, while and the result below the line. Next to the line is th



\subsection{Coq}
% Brief overview of proof assistants, reason for the choice of Coq
There are a number of proof assistants available as destination for Ott, out of which Coq and Isabelle provide good extr

Coq is formal proof assistant with a mathematical higher-level language called \textit{Gallina}, based around the Calcul
% Short guide to reading Coq

Objects in Coq can divided into two sorts, \textit{Prop} (propositions) and \textit{Type}. A proposition like $ \forall

\begin{lstlisting}[language={Coq},caption={Coq Prop logic example}]
forall A B : Prop, A /\ B -> B \/ B
```

```
\end{lstlisting}

Predicates like equality and other sets can be used as well

\begin{lstlisting}[language={Coq},caption={Coq Prop predicate example}]
forall x y : Z, x * y = 0 -> x = 0 \/ y = 0
\end{lstlisting}

New predicates can be defined inductively

\begin{lstlisting}[language={Coq},caption={Coq Prop new predicate example}]
 Inductive even : N -> Prop :=
  | even_0 : even 0
  | even_S n : odd n -> even (n + 1)
  with odd : N -> Prop :=
  | odd_S n : even n -> odd (n + 1).
\end{lstlisting}

Data structures can also be defined both inductively and coinductively

\begin{lstlisting}[language={Coq},caption={Coq inductive data structure example}]
 Inductive seq : nat -> Set :=
  | niln : seq 0
  | consn : forall n : nat, nat -> seq n -> seq (S n).
\end{lstlisting}

\begin{lstlisting}[language={Coq},caption={Coq coinductive data structure example}]
 CoInductive stream (A:Type) : Type :=
  | Cons : A -> stream -> stream.
\end{lstlisting}

Functions over these data structures are defined as fixpoints and cofixpoints respectively.

\begin{lstlisting}[language={Coq},caption={Coq fixpoint example}]
 Fixpoint length (n : nat) (s : seq n) {struct s} : nat :=
     match s with
     | niln => 0
     | consn i _ s’ => S (length i s’)
     end.
\end{lstlisting}

Finally theorems can be proven with these propositions and structures.

\begin{minipage}{\linewidth}
\begin{lstlisting}[language={Coq},caption={Coq theorem example}]
Theorem length_corr : forall (n : nat) (s : seq n), length n s = n.
  Proof.
    intros n s.

    (* reasoning by induction over s. Then, we have two new goals
       corresponding on the case analysis about s (either it is
       niln or some consn *)
    induction s.

      (* We are in the case where s is void. We can reduce the
         term: length 0 niln *)
      simpl.

      (* We obtain the goal 0 = 0. *)
      trivial.

      (* now, we treat the case s = consn n e s with induction
         hypothesis IHs *)
      simpl.

      (* The induction hypothesis has type length n s = n.
         So we can use it to perform some rewriting in the goal: *)
      rewrite IHs.

      (* Now the goal is the trivial equality: S n = S n *)
      trivial.

    (* Now all sub cases are closed, we perform the ultimate
       step: typing the term built using tactics and save it as
       a witness of the theorem. *)
  Qed.
\end{lstlisting}
\end{minipage}
Each Lemma, Theorem, Example have a name and a statement. The statement is a proposition. This is followed by the proof

% Methods of extraction

Coq also provides built in facilities for the certified extraction of code to OCaml, Haskell and Scheme. These can be in


\begin{minipage}{\linewidth}
\begin{lstlisting}[caption={Coq to OCaml extraction of seq}]
type nat =
| O
| S of nat

type seq =
| Niln
| Consn of nat * nat * seq
\end{lstlisting}
\end{minipage}

\begin{minipage}{\linewidth}
\begin{lstlisting}[caption={Coq to OCaml extraction of length}]
(** val length : nat -> seq -> nat **)

let rec length n = function
| Niln -> O
| Consn (i, n0, s') -> S (length i s')
\end{lstlisting}
\end{minipage}

% Extraction plugin

Out of the box, Coq does not provide facilities for the extraction of so called logical inductive systems. These are ess


\begin{minipage}{\linewidth}
\begin{lstlisting}[language={Coq},caption={Coq logical inductive example}]
Inductive add : nat -> nat -> nat -> Prop :=
| addO : forall n , add n O n
| addS : forall n m p, add n m p -> add n (S m) (S p).
\end{lstlisting}
```

```
\end{minipage}

However with the help of a plugin developed by David Delahaye, Catherine Dubois, Jean-Fr\'ed\'eric \'Etienne and Pierre-Nicola

\subsection{OCaml}
% Brief overview of OCaml
% Short guide to reading OCaml

\cleardoublepage
\chapter{Implementation}
\section{The semantics}
\subsection{Expressions}
\subsubsection{Arrow types}
\subsubsection{Sum types}
\subsubsection{Product types}
\subsubsection{Fixpoint combinator}
\subsubsection{Monadic primitives}
\subsubsection{Fork}
\subsubsection{Computation placeholders}
\subsection{Transition system}
\subsubsection{Labelled transitions}
\subsubsection{Select operator}
\subsection{Type system}
\section{Proof assisstant system}
\subsection{Outline of the proof assistant code}
\subsection{Modification}
\subsection{Extractable version}
\section{OCaml system}
\subsection{Outline of the OCaml code}
\subsection{Hand modifications and justifications}
\subsection{Sugar}



\cleardoublepage
\chapter{Evaluation}

\section{Theoretical evaluation}
Properties to evaluate: monadic laws, fork commutativity and associativity (liveness ?), type preservation and progress ?.

\subsection{Methods}
\subsubsection{Weak bisimilarity}
Intro to weak bisimilarity.


What form does a general weak bisimilarity proof take.


How does it appear here.
\subsection{Properties}
\subsubsection{Monadic laws}
\subsubsection{Fork commutativity}
Outline of fork commutativity
\subsubsection{Fork associativity}
\subsubsection{Liveness?}
\subsubsection{Type preservation?}
\subsubsection{Progress?}
%Theoretical evaluation
% - Monadic laws
% - Fork commutativity
% - Fork associativity (limited?)
% - Liveness ?
% - Type preservation ?
% - Progress

% Practical evaluation
\section{Practical evaluation}
Evaluation of speed and memory requirements absolutely and relative to implementations in LWT and Async.
\subsection{Methods}
\subsection{Examples}
\subsubsection{Kahn process network}
\subsubsection{Eratosthene Sieve}
\subsubsection{Concurrent sort}
%

\cleardoublepage
\chapter{Conclusion}

I hope that this rough guide to writing a dissertation is \LaTeX\ has
been helpful and saved you time.




\cleardoublepage

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% the bibliography

\addcontentsline{toc}{chapter}{Bibliography}
\bibliography{refs}
\cleardoublepage

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% the appendices
\appendix

\chapter{Ott source}

\section{diss.tex}
{\scriptsize\verbatiminput{diss.tex}}

\section{proposal.tex}
{\scriptsize\verbatiminput{proposal.tex}}

\section{propbody.tex}
{\scriptsize\verbatiminput{propbody.tex}}




\cleardoublepage

\chapter{Makefile}

\section{\label{makefile}Makefile}
{\scriptsize\verbatiminput{makefile.txt}}

\section{refs.bib}
```

```
{\scriptsize\verbatiminput{refs.bib}}


\cleardoublepage

\chapter{Project Proposal}

\input{partIIproposal}

\end{document}
```

A.2 proposal.tex

A.3 propbody.tex

Appendix B

Makefile

B.1 Makefile

B.2 refs.bib

```
@MISC{OCaml,
 title = {OCaml},
 howpublished = {\url{http://ocaml.org/}}
}

@MISC{LWT,
 title = {LWT, Lightweight Threading library},
 howpublished = {\url{http://ocsigen.org/lwt/}}
}

@MISC{Async,
 title = {Async, open source concurrency library},
 author =  {Jane Street}
 howpublished = {\url{http://janestreet.github.io/}}
}
@MISC{Lem,
 title = {Lem, a tool for lightweight executable mathematics.},
 howpublished = {\url{http://www.cs.kent.ac.uk/people/staff/sao/lem/}}
}
@MISC{Ott,
  title = {Ott, a tool for writing definitions of programming languages and calculi},
  howpublished = {\url{http://www.cl.cam.ac.uk/~pes20/ott/}},
  author = "Francesco Zappa Nardelli"
  }

@MISC{Claessen99functionalpearls, author = {Koen Claessen}, title = {Functional Pearls: A Poor Man’s Concurrency Monad},

@article{deleuzelight,
  title={Light Weight Concurrency in OCaml: Continuations, Monads, Promises, Events},
  author={Deleuze, Christophe}
}
```

```
@inproceedings{vouillon2008lwt,
  title={Lwt: a cooperative thread library},
  author={Vouillon, J{\'e}r{\^o}me},
  booktitle={Proceedings of the 2008 ACM SIGPLAN workshop on ML},
  pages={3--12},
  year={2008},
  organization={ACM}
}

@MISC{Coq,
  title = {Ott, a tool for writing definitions of programming languages and calculi},
  howpublished = {\url{http://coq.inria.fr/}}}

@MISC{OCamlLightWeb,
    title = {Ott, a tool for writing definitions of programming languages and calculi},
    howpublished = {\url{http://www.cl.cam.ac.uk/~so294/ocaml/}},
    year={2008}}


@incollection{OCamlLight,
  title={A sound semantics for OCaml light},
  author={Owens, Scott},
  booktitle={Programming Languages and Systems},
  pages={1--15},
  year={2008},
  publisher={Springer}
}

@article{benton2008mechanized,
  title={A mechanized bisimulation for the nu-calculus},
  author={Benton, Nick and Koutavas, Vasileios},
  journal={Higher-Order and Symbolic Computation (to appear, 2013)},
  year={2008}
}

@article{blazy2009mechanized,
  title={Mechanized semantics for the Clight subset of the C language},
  author={Blazy, Sandrine and Leroy, Xavier},
  journal={Journal of Automated Reasoning},
  volume={43},
  number={3},
  pages={263--288},
  year={2009},
  publisher={Springer}
}
@incollection{blazy2006formal,
  title={Formal verification of a C compiler front-end},
  author={Blazy, Sandrine and Dargaye, Zaynah and Leroy, Xavier},
  booktitle={FM 2006: Formal Methods},
  pages={460--475},
  year={2006},
  publisher={Springer}
}
@article{leroy2009formal,
  title={Formal verification of a realistic compiler},
  author={Leroy, Xavier},
  journal={Communications of the ACM},
  volume={52},
  number={7},
  pages={107--115},
  year={2009},
  publisher={ACM}
}

@incollection{letouzey2008extraction,
  title={Extraction in coq: An overview},
  author={Letouzey, Pierre},
  booktitle={Logic and Theory of Algorithms},
  pages={359--369},
  year={2008},
  publisher={Springer}
}
@incollection{berger1993program,
  title={Program extraction from normalization proofs},
  author={Berger, Ulrich},
  booktitle={Typed Lambda Calculi and Applications},
  pages={91--106},
  year={1993},
  publisher={Springer}
}
@inproceedings{berger1995program,
  title={Program extraction from classical proofs},
  author={Berger, Ulrich and Schwichtenberg, Helmut},
  booktitle={Logic and Computational Complexity},
  pages={77--97},
  year={1995},
  organization={Springer}
}
@incollection{letouzey2003new,
  title={A new extraction for Coq},
  author={Letouzey, Pierre},
  booktitle={Types for proofs and programs},
  pages={200--219},
  year={2003},
  publisher={Springer}
}
@incollection{delahaye2007extracting,
  title={Extracting purely functional contents from logical inductive types},
  author={Delahaye, David and Dubois, Catherine and {\'E}tienne, Jean-Fr{\'e}d{\'e}ric},
  booktitle={Theorem Proving in Higher Order Logics},
  pages={70--85},
  year={2007},
  publisher={Springer}
}

@incollection{fournet2003jocaml,
  title={JoCaml: A language for concurrent distributed and mobile programming},
  author={Fournet, C{\'e}dric and Le Fessant, Fabrice and Maranget, Luc and Schmitt, Alan},
  booktitle={Advanced Functional Programming},
  pages={129--158},
  year={2003},
  publisher={Springer}
}

@inproceedings{shan2004shift,
  title={Shift to control},
  author={Shan, Chung-chieh},
  booktitle={Proceedings of the 5th workshop on Scheme and Functional Programming},
  pages={99--107},
  year={2004}
}

@inproceedings{friedman1988applications,
  title={Applications of continuations},
  author={Friedman, Daniel P},
  booktitle={Proceedings of the ACM Conference on Principles of Programming Languages},
  year={1988}
}

@incollection{kiselyov2010delimited,
  title={Delimited control in OCaml, abstractly and concretely: System description},
  author={Kiselyov, Oleg},
  booktitle={Functional and Logic Programming},
  pages={304--320},
  year={2010},
  publisher={Springer}
}

@inproceedings{ganz1999trampolined,
  title={Trampolined style},
  author={Ganz, Steven E and Friedman, Daniel P and Wand, Mitchell},
  booktitle={ACM SIGPLAN Notices},
  volume={34},
  number={9},
  pages={18--27},
  year={1999},
  organization={ACM}
}

@article{hoareetal2001tackling,
  title={Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell},
  author={Hoareetal, CAR},
  journal={Engineering theories of software construction},
  volume={180},
  pages={47},
  year={2001},
  publisher={IOS Press}
}


@book{liskov1988promises,
  title={Promises: linguistic support for efficient asynchronous procedure calls in distributed systems},
  author={Liskov, Barbara and Shrira, Liuba},
  volume={23},
  number={7},
  year={1988},
  publisher={ACM}
}


@MISC{Ocamlnet,
 title = {OCamlNet},
 howpublished = {\url{http://projects.camlcity.org/projects/ocamlnet.html}},
 author = {Gerd Stolpmann}
}

@inproceedings{sevvcik2011relaxed,
  title={Relaxed-memory concurrency and verified compilation},
  author={{\^S}ev{\v{c}}ik, Jaroslav and Vafeiadis, Viktor and Zappa Nardelli, Francesco and Jagannathan, Suresh and Sewe
  booktitle={ACM SIGPLAN Notices},
  volume={46},
  number={1},
  pages={43--54},
  year={2011},
  organization={ACM}
}

@book{lochbihler2012machine,
  title={A Machine-Checked, Type-Safe Model of Java Concurrency: Language, Virtual Machine, Memory Model, and Verified Co
  author={Lochbihler, Andreas},
  year={2012},
  publisher={KIT Scientific Publishing}
}


@MISC{functory,
title = {Functory},
howpublished = {\url{https://www.lri.fr/~filliatr/functory/About.html}},
urldate = {2014-03-20},
author={Kalyanasundaram, Jean-Christophe Filli{\^a}tre K},
year={2010}
}

@misc{ocamlmpi,
  title={OCamlMPI: Interface with the MPI Message-passing Interface},
  author={Leroy, Xavier}
}



@MISC{jocaml,
title = {The {JoCaml} system},
 author = {Mandel, Louis and Maranget, Luc},
year = {2007},
howpublished = {\url{http://jocaml.inria.fr/}},
urldate = {2014-03-20},
}

@incollection{tollitte2012producing,
  title={Producing certified functional code from inductive specifications},
  author={Tollitte, Pierre-Nicolas and Delahaye, David and Dubois, Catherine},
  booktitle={Certified Programs and Proofs},
  pages={76--91},
  year={2012},
  publisher={Springer}
}
```

Appendix C

Project Proposal

C.1 Introduction of work to be undertaken

With the rise of ubiquitous multiple core systems it is necessary for a working programmer to use concurrency to the greatest extent. However concurrent code has never been easy to write as human reasoning is often poorly equipped with the tools necessary to think about such systems. That is why it is essential for a programming language to provide safe and sound primitives to tackle this problem.

My project aims to do this in the OCaml[3] language by developing a lightweight cooperating threading framework that holds correctness as a core value. The functional nature allows the use of one of the most recent trends in languages popular in academia, monads, to be used for a correct implementation.

There have been two very successful frameworks, LWT[2] and Async[27] that both provided the primitives for concurrent development in OCaml however neither is supported by a clear semantic description as their main focus was ease of use and speed.

C.2 Description of starting point

My personal starting points are the courses ML under Windows (IA), Semantics of Programming Languages (IB), Logic and Proof (IB) and Concepts in Programming Languages (IB). Furthermore I have done extracurricular reading into semantics and typing and attended the Denotational Semantics (II) course in the past year.

The preparatory research period has to include familiarising myself with OCaml and the chosen specification and proof assistant tools.

C.3 Substance and structure

The project will consist of first creating a formal specification for a simple monad that has three main operations bind, return and choose. The behaviour of these operations will be specified in a current semantics tool like Lem[1] or Ott[22].

As large amount of research has gone into both monadic concurrency and implementations in OCaml, the project will draw inspiration from Claessen[9], Deleuze[10] and Vouillon[29].

Some atomic, blocking operations will also be specified including reading and writing to a console prompt or file to better illustrate the concurrency properties and make testing and evaluation possible.

This theory driven executable specification will be paired by a hand implementation and will be thoroughly checked against each other to ensure that both adhere to the desired semantics.

Both of these implementations will be then compared against the two current frameworks for simplicity and speed on various test cases.

If time allows, an extension will also be carried out on the theorem prover version of the specification to formally verify that the implementation is correct.

C.4 Criteria

For the project to be deemed a success the following items must be successfully completed.

1. A specification for a monadic concurrency framework must be designed in the format of a semantics tool.
2. This specification needs to be exported to a proof assistant and has a runnable OCaml version
3. Test cases must be written that can thoroughly check a concurrency framework
4. A hand implementation needs to be designed, implemented and tested against the specification
5. The implementations must be compared to the frameworks LWT and Async based on speed
6. The dissertation must be planned and written

In case the extension will also become viable then its success criterion is that there is a clear formal verification accompanying the automated theorem prover version of the specification.

C.5 Timetable

The project will be split into two week packages

C.5.1 Week 1 and 2

Preparatory reading and research into tools that can be used for writing the specification and in the extension, the proofs. The tools of choice at the time of proposal are Ott for the specification step and Coq[4] as the proof assistant. Potentially a meeting arranged in the Computer Lab by an expert in using these tools.

Deliverable: Small example specifications to try out the tool chain, including SKI combinator calculus.

C.5.2 Week 3 and 4

Investigating the two current libraries and their design decisions and planning the necessary parts of specification. Identifying the test cases that are thorough and common in concurrent code.

Deliverable: A document describing the major design decisions of the two libraries, the difference in design of the specification and a set of test cases much like the ones used in OCaml Light [23, 5], but with a concurrency focus.

C.5.3 Week 5 and 6

Writing the specification and exporting to automated theorem provers and OCaml.

Deliverable: The specification document in the format of the semantics tool and exported in the formats of the proof assistant and OCaml.

C.5.4 Week 7 and 8

Hand implement a version that adheres to the specification and test it against the runnable semantics.

C.5.5 Week 9 and 10

Evaluating the implementations of the concurrency framework against LWT and Async. Writing up the halfway report.

Deliverable: Evaluation data and charts, the halfway report.

C.5.6 Week 11 and 12

If unexpected complexity occurs these two weeks can be used to compensate, otherwise starting on the verification proof in the proof assistant.

C.5.7 Week 13 and 14

If necessary adding more primitives (I/O, network) to test with, improving performance and finishing the verification proof. If time allows writing guide for future use of the framework.

C.5.8 Week 15 and 16

Combining all previously delivered documents as a starting point for the dissertation and doing any necessary further evaluation and extension. Creating the first, rough draft of the dissertation.

C.5.9 Week 17 and 18

Getting to the final structure but not necessarily final wording of the dissertation, acquiring all necessary graphs and charts, incorporating ongoing feedback from the supervisor.

C.5.10 Week 19 and 20

Finalising the dissertation and incorporating all feedback and polishing.

C.6 Resource Declaration

The project will need the following resources:

- MCS computer access that is provided for all projects
- The OCaml core libraries and compiler

- The LWT and Async libraries
- The Lem tool
- The Ott tool
- The use of my personal laptop, to work more efficiently

As my personal laptop is included a suitable back-up plan is necessary which will consist of the following:

- A backup to my personal Dropbox account
- A Git repository on Github
- Frequent backups (potentially remotely) to the MCS partition

My supervisor and on request my overseers will receive access to both the Dropbox account and Github repository to allow full transparency.