Tamás Kispéter

Monadic Concurrency in OCaml

Part II in Computer Science

Churchill College

April 27, 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 27, 2014

Contents

List of Figures

Listings

Acknowledgements

Chapter 1

Introduction

This dissertation describes a project to build a concurrency framework for OCaml. This framework is designed with correctness in mind: developing the well defined semantics, modelled in a proof assistant and finally extracted to actual code. The 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[26], C++ and Java[21], 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 common in many applications in computer science with the rise of faster systems often with multiple cores. Concurrency in a computation can be exploited on several 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. Lightweight concurrency often provides faster switch between threads but some blocking operations on the process level will block all internal threads. The threads in this approach expose the points of possible interleaving and the scheduling is done in software.

Most general-purpose languages offer some way of exploiting concurrency in computations. Functional programming is a good fit for concurrency, since it discourages the use of mutable data structures that lead to race conditions. However, support for concurrency in functional languages is often lacking. Functional languages that have both actual industrial applications and large sets of features are of particular interest. These languages include OCaml and Haskell. I focused on OCaml.

1.3 Current implementations of a concurrency framework in OCaml

There are two very successful monadic concurrency frameworks for OCaml. LWT[2] and Async[29]. They both provide the primitives and syntax extensions for concurrent development. Neither is supported by a clear semantic description, because their main focus is ease of use and speed .

LWT, the lightweight cooperative threading library[31] 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. LWT includes many primitives to provide a feature rich framework, including primitives for thread creation, composition and cancellation, thread local storage and support for various synchronisation techniques.

```
1  open Lwt
2
3  let main () =
4    let heads =
5      Lwt_unix.sleep 1.0 >>
6      return (print_endline "Heads");
7    in
8    let tails =
9      Lwt_unix.sleep 2.0 >>
10     return (print_endline "Tails");
11   in
12   lwt () = heads <&> tails in
13   return (print_endline "Finished")
14
15 let _ = Lwt_main.run (main ())
```

Listing 1.1: LWT example

In Listing 1.1 we can see some of the syntax of LWT. Lines 4–6 define heads, a function that sleeps for 1 second and then prints "Head", and lines 8–10 define tails which sleeps for 2 seconds and then prints "Tails". Lines 12–13 create a thread that waits on heads and tails and then prints "Finished". In LWT, semantics mostly follow the principle of continuations. We build a sequence of computations and the scheduler can pick between parallel computations at points of sequencing.

An other implementation, Async is an open source concurrency library for OCaml developed by Jane Street. Unlike LWT the basic semantics are designed with promises in mind. A promise is a container that can be used in place of a value of the same type, but computations with a promise only evaluate when the actual value has been calculated. The concurrency arises naturally by interleaving the fulfilment of these containers.

```
open Core.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 Listing 1.2 we define `heads` and `tails` as Deferred values of the respective code sequences. A Deferred is an implementation of a promise.

There are a number of other experimental implementations of concurrency in OCaml. For example JoCaml[22] implements join calculus over OCaml, Functory[15] focuses on distributed computation, OCamlNet exploits multiple cores and OCamlMPI[17] provides bindings for the standard MPI message passing framework.

1.4 Semantics of concurrency

There has been a lot of work on formulating the semantics of concurrent and distributed systems. Some of the most common models for lightweight concurrency[11] are captured[12] and delimited[16] continuations[27], trampolined style[13], continuation monads[9], promise monads[20] and event based programming (as used, for example in the OCamlNet[28] project). This work focuses on the continuation monad style.

A monad[14] 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 semicolon of imperative programming) or conditional execution. Two operations commonly called bind and return and a type constructor of a parametric type, like $\alpha\, M$ where α is any type, form a monad when they obey a set of axioms called monadic laws.

Most monads support further operations and a concurrency monad is one such monad. Beside the two necessary operations (return and bind) a concurrency monad has to support at least one that deals with concurrent execution. This operation can come in many forms and under many names, for example fork, join or choose. Each with differing signatures and semantics:

- Fork would commonly take two different computations and evaluate them together. Its return semantics would be to return when one thread finished but include the partially completed other computation if possible.
- Join may take many threads, but it waits for all threads to finish.
- Choose can also take many computations, however it would commonly either only evaluate one thread or discard every thread but the one that finished first.

1.5 Semantics to logic

The semantics of concurrency can be modelled in logic, in particular logics used by proof assistants. The developer can use this model to formally verify properties about the semantics[6, 8, 7, 18]. Coq[4], HOL and Isabelle are widely used proof assistants. Tools like Ott[23] help with the modelling process with ascii-art notation and translation to proof assistants and LaTeX.

1.6 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, not necessarily constructive relations. This representation is more amenable to proofs than to actual execution, because there is no need for an input-output relationship. Without this strict requirement on the relation the representation can be more succinct, but hard to extract. Letouzey[19] has shown that many such definitions can be extracted into executable OCaml or Haskell code. Coq and Isabelle provide tools for this extraction. The tools also generate a proof that the extracted 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 concurrent semantics

Concurrency may be modelled in many ways. A popular way of modelling concurrency is with a process calculus. A process calculus is an algebra of processes or threads where. A thread is a unit of control, sometimes also a unit of resources. This algebra often comes with a number of operations like

- $P \mid Q$ for parallel composition where P and Q are processes
- $a.P$ for sequential composition where a is an atomic action and P is a process executed sequentially
- $!P$ for replication where P is a process and $!P \equiv P \mid !P$
- $x\langle y\rangle \cdot P$ and $x(v) \cdot Q$ for sending and receiving messages through channel x respectively

This project aimed to have simple but powerful operational semantics. Simplicity is required in both the design and the interface. There is a short and limited timespan for implementation and an even shorter period for the user to understand the system. On the other hand, the model should have comparable formal properties to full, well known process calculi.

I focused on providing primitives for operations on processes including parallel and sequential composition and recursion. Formal treatment of communication channels have been left out to limit the scope of the project.

2.2 Choice of implementation style

Deleuze[11] surveyed a number of implementation styles of lightweight concurrency for OCaml. The styles fall in two broad categories: direct and indirect styles. Direct styles like captured and delimited continuations involve keeping an explicit queue of continuations that can be executed at any given time and a scheduler that picks the next element from the queue. Indirect styles include the trampolined style and two monadic styles: continuations and promises.

Simplicity and similarity to current implementations like LWT and Async were the two factors in the decision between these styles. Both direct styles and the promise monad style keep concurrency state data that is external to the language and has to be maintained at runtime explicitly. The extra structure would make the implementation slightly more complex. LWT and Async both provide monadic style interfaces therefore I chose the continuation monad style.

2.3 Design of monadic semantics

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 property 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:

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

   That is to say wrapping and then subsequently unwrapping behaves as an identity.

This description entails three operations (lift, join and map) 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.

Most implementations go along the following lines: there is a parametric type $\mathbf{con}\,\alpha$ where α is the type parameter. Note **con** is an arbitrary name, marker for a particular monad. The I/O monad would have IO as the marker.

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

The bind (often written as $\gg\!=$) takes a monadic value (that is one in the parametric type $\mathbf{con}\,\alpha$) and a function that can map the inner value to a new monadic value (that is, it has type $\alpha \to \mathbf{con}\,\beta$). Bind then returns a $\mathbf{con}\,\beta$. With types $\gg\!=$ means: $\gg\!= : \forall\alpha\,\beta.\ \mathbf{con}\,\alpha \to (\alpha \to \mathbf{con}\,\beta) \to \mathbf{con}\,\beta$.

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 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.

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.

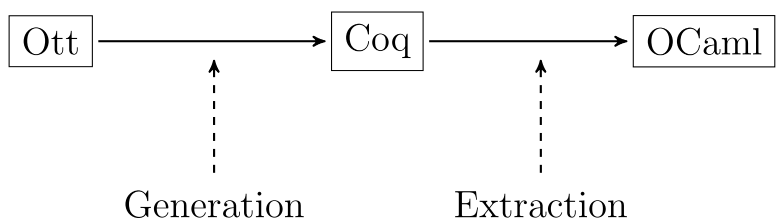

Figure 2.1: High level view of the tool chain

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. The Ott 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 Listing 2.2 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

In Listing 2.3 v is a subgrammar of t. The statement v <:: t is exported as a target language subroutine that checks whether the value relation holds and during translation Ott checks for obvious bugs.

Another common feature of semantics is substitution of values for variables, for example in function application. Substitution 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 in Listing 2.4 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 Listing 2.5 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 destinations 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 $\forall A, B.\ A \wedge B \rightarrow B \vee B$ translates to the snippet in Listing 2.7.

```
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 [10, 30] by marking different modalities of the inductively generated proposition we can generate code with an input-output convention.

```
(** val add12 : nat → nat → nat **)

let rec add12 p1 p2 =
  match (p1, p2) with
  | (n, O) → n
  | (n, S m) →
    (match add12 n m with
     | p → S p
     | _ → assert false (*  *))
  | _ → assert false (*  *)
```

Listing 2.17: Coq to OCaml extraction of a logical inductive relation

Most descriptions of reduction relations and indeed the output of Ott is of this kind, therefore this plugin helps with the extraction of a reduction relation directly.

2.4.3 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.

OCaml, like many other functional languages support a wide range of features, from simple functions, to mutually recursive functions with pattern matching.

```
let square x = x * x
```

Listing 2.18: OCaml simple function example: square

```
let rec fact x =
    if x <= 1 then 1 else x * fact (x - 1)
```

Listing 2.19: OCaml recursive function example: factorial

```
let rec sort = function
    | [] -> []
    | x :: l -> insert x (sort l)
  and insert elem = function
    | [] -> [elem]
    | x :: l -> if elem < x then elem :: x :: l
                else x :: insert elem l
```

Listing 2.20: OCaml complex function example: insertion sort

Furthermore, it was designed as a versatile, general purpose programming language. OCaml features include objects, modules, support for imperative style and higher order functions.

```
let add_polynom p1 p2 =
    let n1 = Array.length p1
    and n2 = Array.length p2 in
    let result = Array.create (max n1 n2) 0 in
    for i = 0 to n1 - 1 do result.(i) <- p1.(i) done;
    for i = 0 to n2 - 1 do result.(i) <- result.(i) + p2.(i) done;
```

Listing 2.21: OCaml imperative function example

```
let rec sigma f = function
    | [] -> 0
    | x :: l -> f x + sigma f l
```

Listing 2.22: OCaml higher order functions example

```
type expression =
    | Num of int
    | Var of string
    | Let of string * expression * expression
    | Binop of string * expression * expression
```

Listing 2.23: OCaml data structure example

```
let rec eval env = function
    | Num i -> i
    | Var x -> List.assoc x env
    | Let (x, e1, in_e2) ->
       let val_x = eval env e1 in
       eval ((x, val_x) :: env) in_e2
    | Binop (op, e1, e2) ->
       let v1 = eval env e1 in
       let v2 = eval env e2 in
       eval_op op v1 v2
  and eval_op op v1 v2 =
    match op with
    | "+" -> v1 + v2
    | "-" -> v1 - v2
    | "*" -> v1 * v2
    | "/" -> v1 / v2
    | _ -> failwith ("Unknown operator: " ^ op)
```

Listing 2.24: OCaml evaluation function example

Chapter 3

Implementation

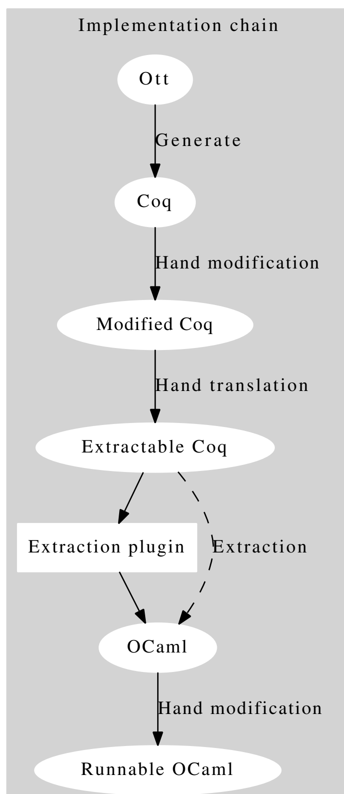

Figure 3.1: Detailed outline of the implementation

3.1 The semantics

efmowfiowfiwe

nofnownfoiwnfoiw

3.1.1 Expressions

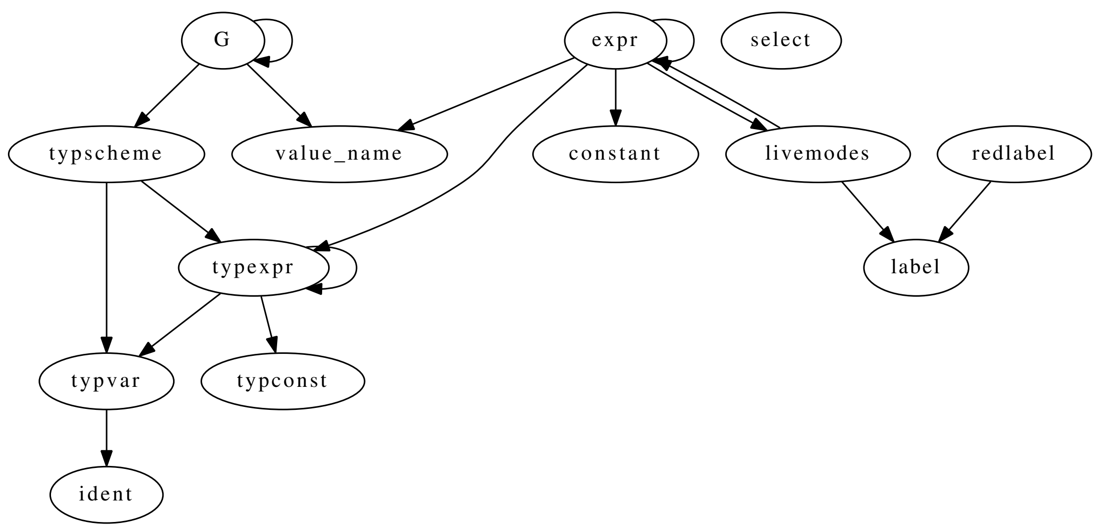

Figure 3.2: Grammar dependencies

Arrow types

In this section I describe the implementation of arrow types defined in this language. The style and details are based on Pierce[25, p. 103].

Sum types

In this section I describe the implementation of sum types defined in this language. The style and details are based on Pierce[25, p. 132].

Product types

In this section I describe the implementation of sum types defined in this language. The style and details are based on Pierce[25, p. 126].

Syntax

e ::=		*terms:*
	x	*value name*
	λx:T.e	*abstraction*
	e e	*application*
v ::=		*values:*
	λx:T.e	*abstraction value*
TS ::=		*typescheme:*
	(TV1, ..., TVn) T	*general type*
T ::=		*types:*
	TV	*type variable*
	T$\rightarrow$T	*$\rightarrow$ type*
Γ ::=		*contexts:*
	$\emptyset$	*empty context*
	Γ, x:TS	*term variable binding*
rl ::=		*labels:*
	τ	*silent*
	l	*action*
s ::=		*select:*
	1	*first*
	2	*second*

Evaluation $\boxed{e \xrightarrow[s]{rl} e'}$

$$\frac{}{(\lambda\, x \,:\, T.e)\, v \xrightarrow[s]{\tau} \{v/x\}e} \quad \text{(R-Subst)}$$

$$\frac{e' \xrightarrow[s]{rl} e''}{e\, e' \xrightarrow[s]{rl} e\, e''} \quad \text{(R-App1)}$$

$$\frac{e \xrightarrow[s]{rl} e'}{e\, v \xrightarrow[s]{rl} e'\, v} \quad \text{(R-App2)}$$

Typing $\boxed{\Gamma \vdash \mathrm{t} : \mathrm{T}}$

$$\frac{\mathrm{x{:}TS} \in \Gamma \qquad TS > T}{\Gamma \vdash \mathrm{x} : \mathrm{T}} \quad \text{(T-Var)}$$

$$\frac{\Gamma \vdash e : T_1 \rightarrow T_2 \qquad \Gamma \vdash e' : T_1}{\Gamma \vdash e\, e' : T_2} \quad \text{(T-App)}$$

$$\frac{\Gamma, x : ()T_1 \vdash e : T_2}{\Gamma \vdash \lambda\, x : T_1.e : T_1 \rightarrow T_2} \quad \text{(T-Fun)}$$

Figure 3.3: Syntax and semantics of arrow types

Fixpoint combinator

In this section I describe the implementation of a fixpoint operator defined in this language. The style and details are based on Pierce[25, p. 144].

Monadic primitives

The concurrency 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 $\alpha \to \alpha\,\mathbf{con}$ 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 $\alpha\,\mathbf{con} \to (\alpha \to \beta\,\mathbf{con}) \to \beta\,\mathbf{con}$, 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.

Fork

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

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

Computation placeholders

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

3.3.2 Hand modifications and justifications

3.3.3 Sugar

Syntax

e ::=	*terms:*
left e	*left*
right e	*right*
Case e_1 of **left** $x_1 \Rightarrow e_2$ | **right** $x_2 \Rightarrow e_3$	*case*
v ::=	*values:*
left v	*left*
right v	*right*
T ::=	*types:*
$T + T$	*sum type*

Evaluation $\boxed{e \xrightarrow[s]{rl} e'}$

$$\frac{e \xrightarrow[s]{rl} e'}{(\text{Case } e \text{ of } \mathbf{left}\ x_1 \Rightarrow e_2 \mid \mathbf{right}\ x_2 \Rightarrow e_3\ ) \xrightarrow[s]{rl} (\text{Case } e' \text{ of } \mathbf{left}\ x_1 \Rightarrow e_2 \mid \mathbf{right}\ x_2 \Rightarrow e_3\ )} \quad \text{(R-Case)}$$

$$\frac{}{(\text{Case } \mathbf{left}\, v \text{ of } \mathbf{left}\ x_1 \Rightarrow e_2 \mid \mathbf{right}\ x_2 \Rightarrow e_3\ ) \xrightarrow[s]{\tau} \{v/x_1\}e_2} \quad \text{(R-CaseL)}$$

$$\frac{}{(\text{Case } \mathbf{right}\, v \text{ of } \mathbf{left}\ x_1 \Rightarrow e_2 \mid \mathbf{right}\ x_2 \Rightarrow e_3\ ) \xrightarrow[s]{\tau} \{v/x_2\}e_3} \quad \text{(R-CaseR)}$$

$$\frac{e \xrightarrow[s]{rl} e'}{\mathbf{left}\, e \xrightarrow[s]{rl} \mathbf{left}\, e'} \quad \text{(R-Left)}$$

$$\frac{e \xrightarrow[s]{rl} e'}{\mathbf{right}\, e \xrightarrow[s]{rl} \mathbf{right}\, e'} \quad \text{(R-Right)}$$

Typing $\boxed{\Gamma \vdash \mathrm{t} : \mathrm{T}}$

$$\frac{\Gamma \vdash\ e : T_1}{\Gamma \vdash \mathbf{left}\, e : T_1 + T_2} \quad \text{(T-Left)}$$

$$\frac{\Gamma \vdash\ e : T_2}{\Gamma \vdash \mathbf{right}\, e : T_1 + T_2} \quad \text{(T-Right)}$$

$$\frac{\Gamma \vdash\ e : T_1 + T_2 \qquad \Gamma, x_1 : ()T_1 \vdash\ e_2 : T \qquad \Gamma, x_2 : ()T_2 \vdash\ e_3 : T}{\Gamma \vdash (\text{Case } e \text{ of } \mathbf{left}\ x_1 \Rightarrow e_2 \mid \mathbf{right}\ x_2 \Rightarrow e_3\ ) : T} \quad \text{(T-Case)}$$

Figure 3.4: Syntax and semantics of sum types

Syntax

e ::=		*terms:*
	$\{e, e'\}$	*pair*
	pair	*to pair*
	proj1	*first projection*
	proj2	*second projection*
v ::=		*values:*
	$\{v, v'\}$	*pair*
	pair	*to pair*
	pair v	*to pair, partial*
	proj1	*first projection*
	proj2	*second projection*
T ::=		*types:*
	$T \star T$	*product type*

Evaluation $\boxed{e \xrightarrow[s]{rl} e'}$

$$\frac{e \xrightarrow[s]{rl} e''}{\{e, e'\} \xrightarrow[s]{rl} \{e'', e'\}} \quad \text{(R-Pair1)}$$

$$\frac{e' \xrightarrow[s]{rl} e''}{\{v, e'\} \xrightarrow[s]{rl} \{v, e''\}} \quad \text{(R-Pair2)}$$

$$\mathbf{pair}\, v\, v' \xrightarrow[s]{\tau} \{v, v'\} \quad \text{(R-InPair)}$$

$$\mathbf{proj1}\, \{v,\ v'\} \xrightarrow[s]{\tau} v \quad \text{(R-Proj1)}$$

$$\mathbf{proj2}\, \{v,\ v'\} \xrightarrow[s]{\tau} v' \quad \text{(R-Proj2)}$$

Typing $\boxed{\Gamma \vdash \mathrm{t} : \mathrm{T}}$

$$\frac{\Gamma \vdash e : T_1 \qquad \Gamma \vdash e' : T_2}{\Gamma \vdash \{e,\ e'\} : T_1 \star T_2} \quad \text{(T-Pair)}$$

$$\Gamma \vdash \mathbf{pair} : T_1 \to T_2 \to T_1 \star T_2 \quad \text{(T-InPair)}$$

$$\Gamma \vdash \mathbf{proj1} : T_1 \star T_2 \to T_1 \quad \text{(T-Proj1)}$$

$$\Gamma \vdash \mathbf{proj2} : T_1 \star T_2 \to T_2 \quad \text{(T-Proj2)}$$

Figure 3.5: Syntax and semantics of product types

Syntax		
e ::=		*terms:*
	fix	*fixpoint*
v ::=		*values:*
	fix	*fixpoint*

Evaluation $\boxed{e \xrightarrow[s]{rl} e'}$

$$\mathbf{fix}\,(\lambda x : T.e) \xrightarrow[s]{\tau} \{(\mathbf{fix}\,(\lambda x : T.e))/x\}e \qquad \text{(R-Fix)}$$

Typing $\boxed{\Gamma \vdash \mathrm{t} : \mathrm{T}}$

$$\Gamma \vdash \mathbf{fix} : (T \to T) \to T \qquad \text{(T-Fix)}$$

Figure 3.6: Syntax and semantics of the fixpoint operator

Syntax

e ::=		*terms:*
	LE	*live expression*
	()	*unit*
	ret	*return*
	e $\gg=$ e'	*bind*
v ::=		*values:*
	LE	*live expression*
	()	*unit*
	ret	*return*
LE ::=	*Live* LM	*live expressions:*
T ::=		*types:*
	con T	*concurrent*
	unit	*unit*
LM ::=		*live modes:*
	expr e	*expression*
	comp l	*computation*

Evaluation $\boxed{e \xrightarrow[s]{rl} e'}$

$$\textbf{ret}\ v \xrightarrow[s]{\tau} \textit{Live}\ \text{expr}\ v \quad \text{(R-Return)}$$

$$\frac{e \xrightarrow[s]{rl} e''}{e \gg= e' \xrightarrow[s]{rl} e'' \gg= e'} \quad \text{(R-Movebind)}$$

$$\textit{Live}\ \text{expr}\ e \gg= e' \xrightarrow[s]{\tau} e'e \quad \text{(R-Dobind)}$$

$$\textit{Live}\ \text{comp}\ l \gg= e' \xrightarrow[s]{l} e'\,() \quad \text{(R-Compbind)}$$

Typing $\boxed{\Gamma \vdash \mathrm{t} : \mathrm{T}}$

$$\frac{\Gamma \vdash e : \textbf{con}\ T_1 \qquad \Gamma \vdash e' : T_1 \rightarrow \textbf{con}\ T_2}{\Gamma \vdash e \gg= e' : \textbf{con}\ T_2} \quad \text{(T-Bind)}$$

$$\Gamma \vdash \textbf{ret} : T \rightarrow \textbf{con}\ T \quad \text{(T-Ret)}$$

$$\Gamma \vdash \textit{Live}\ \text{comp}\ l : \textbf{con unit} \quad \text{(T-LMComp)}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \textit{Live}\ \text{expr}\ e : \textbf{con}\ T} \quad \text{(T-LMExpr)}$$

$$\Gamma \vdash () : \textbf{unit} \quad \text{(T-Unit)}$$

Figure 3.7: Syntax and semantics of the monadic primitives, ret and bind

Evaluation $\boxed{e \xrightarrow[s]{rl} e'}$

$$\frac{e \xrightarrow[s]{rl} e'}{\textbf{fork}\ (\mathit{Live\ expr}\ e)(\mathit{Live}\ lm) \xrightarrow[1]{rl} \textbf{fork}\ (\mathit{Live\ expr}\ e')(\mathit{Live}\ lm)} \quad \text{(R-Forkmove1)}$$

$$\frac{e \xrightarrow[s]{rl} e'}{\textbf{fork}\ (\mathit{Live}\ lm)(\mathit{Live\ expr}\ e) \xrightarrow[2]{rl} \textbf{fork}\ (\mathit{Live}\ lm)(\mathit{Live\ expr}\ e')} \quad \text{(R-Forkmove2)}$$

$$\textbf{fork}\ (\mathit{Live\ expr}\ v)(\mathit{Live}\ lm) \xrightarrow[1]{\tau} \mathit{Live\ expr}\ \textbf{left}\{v, (\mathit{Live}\ lm)\} \quad \text{(R-Forkdeath1)}$$

$$\textbf{fork}\ (\mathit{Live}\ lm)(\mathit{Live\ expr}\ v) \xrightarrow[2]{\tau} \mathit{Live\ expr}\ \textbf{right}\{(\mathit{Live}\ lm), v\} \quad \text{(R-Forkdeath2)}$$

$$\textbf{fork}\ (\mathit{Live\ comp}\ l)(\mathit{Live}\ lm) \xrightarrow[1]{l} \mathit{Live\ expr}\ \textbf{left}\{(), (\mathit{Live}\ lm)\} \quad \text{(R-Forkdocomp1)}$$

$$\textbf{fork}\ (\mathit{Live}\ lm)(\mathit{Live\ comp}\ l) \xrightarrow[2]{l} \mathit{Live\ expr}\ \textbf{right}\{(\mathit{Live}\ lm), ()\} \quad \text{(R-Forkdocomp2)}$$

Syntax

e ::=		*terms:*
	fork	*fork*
v ::=		*values:*
	fork	*fork*
	fork v	*fork, partial*

Typing $\boxed{\Gamma \vdash \mathrm{t} : \mathrm{T}}$

$$\Gamma \vdash \textbf{fork} : \textbf{con}\ T_1 \to \textbf{con}\ T_2 \to \textbf{con}\ ((T_1 \star \textbf{con}\ T_2) + ((\textbf{con}\ T_1) \star T_2) \quad \text{(T-Fork)}$$

Figure 3.8: Syntax and semantics of the fork operator

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] David Delahaye, Catherine Dubois, and Jean-Frédéric Étienne. Extracting purely functional contents from logical inductive types. In *Theorem Proving in Higher Order Logics*, pages 70–85. Springer, 2007.

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

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

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

[14] 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.

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

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

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

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

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

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

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

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

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

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

[25] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[26] 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.

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

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

[29] Jane Street. Async, open source concurrency library. `http://janestreet.github.io/`.

[30] 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.

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

Appendix A

Full semantics

A.1 diss.tex

$value\_name,\ x$			
$label,\ lab$			
$ident$			
$index,\ i,\ j,\ n,\ m$			
$typvar,\ tv$	$::=$		
	$\vert$	$'ident$	
$typconst,\ tc$	$::=$		
	$\vert$	$\mathbf{tunit}$	
$typexpr,\ t$	$::=$		
	$\vert$	$typconst$	
	$\vert$	$typvar$	
	$\vert$	$typexpr \to typexpr'$	
	$\vert$	$typexpr * typexpr'$	
	$\vert$	$\mathbf{con}\ typexpr$	
	$\vert$	$typexpr + typexpr'$	
	$\vert$	$(typexpr)$	S
$typscheme,\ ts$	$::=$		
	$\vert$	$(typvar_1, .., typvar_n) typexpr$	bind $typvar_1..typvar_n$ in $typexpr$
	$\vert$	$\mathbf{generalise}\,(\Gamma, t)$	M
$constant,\ c$	$::=$		
	$\vert$	$\mathbf{ret}$	
	$\vert$	$\mathbf{fork}$	
	$\vert$	$\mathbf{unit}$	
	$\vert$	$\mathbf{stop}$	
	$\vert$	$\mathbf{pair}$	
	$\vert$	$proj1$	
	$\vert$	$proj2$	
$redlabel,\ rl$	$::=$		
	$\vert$	τ	
	$\vert$	lab	
$livemodes,\ lm$	$::=$		
	$\vert$	$comp\ lab$	
	$\vert$	$exp\ expr$	
	$\vert$	(lm)	S
$expr,\ e$	$::=$		
	$\vert$	$value\_name$	
	$\vert$	$constant$	

	|	$expr\ expr'$	
	|	$expr >>= expr'$	
	|	**function** $value\_name : typexpr \rightarrow expr$	bind $value\_name$ in $expr$
	|	**fix** e	
	|	$Live\ lm$	
	|	$\{e, e'\}$	
	|	$(expr)$	S
	|	**left** e	
	|	**right** e	
	|	$Case\ e_1\ of$ **left** $x_1 \Rightarrow e_2 \vert$ **right** $x_2 \Rightarrow e_3$	
	|	$\{v/x\}e$	M
	|	$\{($**fix** $(functionx : t \rightarrow e))/x'\}e$	M
$value,\ v$	::=		
	|	$constant$	
	|	**function** $value\_name : typexpr \rightarrow expr$	
	|	$Live\ lm$	
	|	**left** v	
	|	**right** v	
	|	$\{v, v'\}$	
	|	(v)	S
$select,\ s$	::=		
	|	1	
	|	2	
Γ	::=		
	|	**empty**	
	|	$\Gamma, value\_name : typscheme$	
$formula$	::=		
	|	$judgement$	
	|	**not** $(formula)$	
	|	$typscheme > t$	
	|	$typscheme = typscheme'$	
	|	$value\_name = value\_name'$	
$terminals$	::=		
	|	$\rightarrow$	
	|	**function**	
	|	$\vdash$	
	|	$\longrightarrow$	

$\mid$	$\{$
$\mid$	$\}$
$\mid$	$[$
$\mid$	$]$
$\mid$	**con**
$\mid$	$comp$
$\mid$	exp
$\mid$	$Live$
$\mid$	**fix**
$\mid$	τ
$\mid$	$>>=$
$\mid$	$*$
$\mid$	$,$
$\mid$	**left**
$\mid$	**right**
$\mid$	$Case$
$\mid$	of
$\mid$	$\Rightarrow$
$\mid$	$+$
$\mid$	$:$

$Jtype$	$::=$	
	$\mid$	$value\_name : typscheme\, \mathbf{in}\, \Gamma$
	$\mid$	$\Gamma \vdash constant : t$
	$\mid$	$\Gamma \vdash e : t$

Jop	$::=$	
	$\mid$	$e \xrightarrow[s]{rl} e'$

$judgement$	$::=$	
	$\mid$	$Jtype$
	$\mid$	Jop

$user\_syntax$	$::=$	
	$\mid$	$value\_name$
	$\mid$	$label$
	$\mid$	$ident$
	$\mid$	$index$
	$\mid$	$typvar$
	$\mid$	$typconst$
	$\mid$	$typexpr$
	$\mid$	$typscheme$
	$\mid$	$constant$

$|$ *redlabel*
$|$ *livemodes*
$|$ *expr*
$|$ *value*
$|$ *select*
$|$ Γ
$|$ *formula*
$|$ *terminals*

$\boxed{\mathit{value\_name} : \mathit{typscheme}\ \mathbf{in}\ \Gamma}$

$$\frac{}{\mathit{value\_name} : \mathit{typscheme}\ \mathbf{in}\ \Gamma, \mathit{value\_name} : \mathit{typscheme}} \quad \text{VTSIN\_VN1}$$

$$\frac{\begin{array}{c}\mathit{value\_name} : \mathit{typscheme}\ \mathbf{in}\ \Gamma \\ \mathbf{not}\ (\mathit{value\_name} = \mathit{value\_name}')\end{array}}{\mathit{value\_name} : \mathit{typscheme}\ \mathbf{in}\ \Gamma, \mathit{value\_name}' : \mathit{typscheme}'} \quad \text{VTSIN\_VN2}$$

$\boxed{\Gamma \vdash \mathit{constant} : t}$

$$\frac{}{\Gamma \vdash \mathbf{ret} : t \to \mathbf{con}\ t} \quad \text{CONSTANT\_RET}$$

$$\frac{}{\Gamma \vdash \mathbf{fork} : (\mathbf{con}\ t_1) \to ((\mathbf{con}\ t_2) \to (\mathbf{con}\ ((t_1 * (\mathbf{con}\ t_2)) + ((\mathbf{con}\ t_1) * t_2))))} \quad \text{CONSTANT\_FORK}$$

$$\frac{}{\Gamma \vdash \mathbf{unit} : \mathbf{tunit}} \quad \text{CONSTANT\_UNIT}$$

$$\frac{}{\Gamma \vdash \mathbf{stop} : t} \quad \text{CONSTANT\_STOP}$$

$$\frac{}{\Gamma \vdash \mathbf{pair} : t_1 \to (t_2 \to (t_1 * t_2))} \quad \text{CONSTANT\_PAIR}$$

$$\frac{}{\Gamma \vdash \mathit{proj}1 : (t_1 * t_2) \to t_1} \quad \text{CONSTANT\_PROJ1}$$

$$\frac{}{\Gamma \vdash \mathit{proj}2 : (t_1 * t_2) \to t_2} \quad \text{CONSTANT\_PROJ2}$$

$\boxed{\Gamma \vdash e : t}$

$$\frac{\begin{array}{l}x : \mathit{typscheme}\ \mathbf{in}\ \Gamma \\ \mathit{typscheme} > t\end{array}}{\Gamma \vdash x : t} \quad \text{GET\_VALUE\_NAME}$$

$$\frac{\Gamma \vdash \mathit{constant} : t}{\Gamma \vdash \mathit{constant} : t} \quad \text{GET\_CONSTANT}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : t_1 \to t_2 \\ \Gamma \vdash e' : t_1\end{array}}{\Gamma \vdash e\ e' : t_2} \quad \text{GET\_APPLY}$$

$$\frac{\Gamma, x_1 : (\,)t_1 \vdash e : t}{\Gamma \vdash \mathbf{function}\ x_1 : t_1 \to e : t_1 \to t} \quad \text{GET\_LAMBDA}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathit{Live\,exp}\ e : \mathbf{con}\ t} \quad \text{GET\_LIVE\_EXP}$$

$$\frac{}{\Gamma \vdash \mathit{Live}\ (\mathit{comp\ lab}) : \mathbf{con}\ \mathbf{tunit}} \quad \text{GET\_LIVE\_COMP}$$

$$\frac{\Gamma \vdash e : t \to t}{\Gamma \vdash \mathbf{fix}\, e : t} \quad \text{GET\_FIX}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathbf{con}\, t \\ \Gamma \vdash e' : t \to \mathbf{con}\, t'\end{array}}{\Gamma \vdash e >>= e' : \mathbf{con}\, t'} \quad \text{GET\_BIND}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : t_1 \\ \Gamma \vdash e' : t_2\end{array}}{\Gamma \vdash \{e, e'\} : (t_1 * t_2)} \quad \text{GET\_PAIR}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathbf{left}\, e : t + t'} \quad \text{GET\_TINL}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathbf{right}\, e : t' + t} \quad \text{GET\_TINR}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : t + t' \\ \Gamma, x : (\,)t \vdash e' : t'' \\ \Gamma, x' : (\,)t' \vdash e'' : t''\end{array}}{\Gamma \vdash Case\, e\, of\, \mathbf{left}\, x \Rightarrow e' |\, \mathbf{right}\, x' \Rightarrow e'' : t''} \quad \text{GET\_TCASE}$$

$$\boxed{e \xrightarrow[s]{rl} e'}$$

$$\frac{}{(\mathbf{function}\, x : t \to e)\; v \xrightarrow[s]{\tau} \{v/x\}e} \quad \text{JO\_RED\_APP}$$

$$\frac{e \xrightarrow[s]{rl} e''}{(\mathbf{fork}\,(Live\, exp\, e))\,(Live\, lm) \xrightarrow[1]{rl} (\mathbf{fork}\,(Live\, exp\, e''))\,(Live\, lm)} \quad \text{JO\_RED\_FORKMOVE1}$$

$$\frac{e' \xrightarrow[s]{rl} e''}{(\mathbf{fork}\,(Live\, lm))\,(Live\, exp\, e') \xrightarrow[2]{rl} (\mathbf{fork}\,(Live\, lm))\,(Live\, exp\, e'')} \quad \text{JO\_RED\_FORKMOVE2}$$

$$\frac{}{(\mathbf{fork}\,(Live\, exp\, v))\,(Live\, lm) \xrightarrow[1]{\tau} Live\, exp\,(\mathbf{left}\,((\{v, (Live\, lm)\})))} \quad \text{JO\_RED\_FORKDEATH1}$$

$$\frac{}{(\mathbf{fork}\,(Live\, lm))\,(Live\, exp\, v') \xrightarrow[2]{\tau} Live\, exp\,(\mathbf{right}\,(\{(Live\, lm), v'\}))} \quad \text{JO\_RED\_FORKDEATH2}$$

$$\frac{}{(\mathbf{fork}\,(Live\,(comp\, lab)))\,(Live\, lm) \xrightarrow[1]{lab} Live\, exp\,(\mathbf{left}\,((\{\mathbf{unit}, (Live\, lm)\})))} \quad \text{JO\_RED\_FORKDOCOMP1}$$

$$\frac{}{(\mathbf{fork}\,(Live\, lm))\,(Live\,(comp\, lab)) \xrightarrow[2]{lab} Live\, exp\,(\mathbf{right}\,((\{(Live\, lm), \mathbf{unit}\})))} \quad \text{JO\_RED\_FORKDOCOMP2}$$

$$\frac{}{\mathbf{ret}\, v \xrightarrow[s]{\tau} (Live\, exp\, v)} \quad \text{JO\_RED\_RET}$$

$$\frac{e \xrightarrow[s]{rl} e'}{e >>= e'' \xrightarrow[s]{rl} e' >>= e''} \quad \text{JO\_RED\_EVALBIND}$$

$$\frac{e \xrightarrow[s]{rl} e'}{(Live\, exp\, e) >>= e'' \xrightarrow[s]{rl} (Live\, exp\, e') >>= e''} \quad \text{JO\_RED\_MOVEBIND}$$

$$\frac{}{(Live\,(comp\,lab))\ >>=\ e\ \xrightarrow[s]{lab}\ e\ \mathbf{unit}}\quad \text{JO\_RED\_COMPBIND}$$

$$\frac{}{(Live\,exp\,v)\ >>=\ e\ \xrightarrow[s]{\tau}\ e\ v}\quad \text{JO\_RED\_DOBIND}$$

$$\frac{e'\ \xrightarrow[s]{rl}\ e''}{e\ e'\ \xrightarrow[s]{rl}\ e\ e''}\quad \text{JO\_RED\_CONTEXT\_APP1}$$

$$\frac{e\ \xrightarrow[s]{rl}\ e'}{e\ v\ \xrightarrow[s]{rl}\ e'\ v}\quad \text{JO\_RED\_CONTEXT\_APP2}$$

$$\frac{e\ \xrightarrow[s]{rl}\ e'}{(\mathbf{fix}\ e)\ \xrightarrow[s]{rl}\ (\mathbf{fix}\ e')}\quad \text{JO\_RED\_FIX\_MOVE}$$

$$\frac{}{(\mathbf{fix}\,(\mathbf{function}\,x : t \to e))\ \xrightarrow[s]{\tau}\ \{(\ \mathbf{fix}\,(function x : t \to e))/x\}e}\quad \text{JO\_RED\_FIX\_APP}$$

$$\frac{e\ \xrightarrow[s]{rl}\ e'}{\{e, e''\}\ \xrightarrow[s]{rl}\ \{e', e''\}}\quad \text{JO\_RED\_PAIR\_1}$$

$$\frac{e\ \xrightarrow[s]{rl}\ e'}{\{v, e\}\ \xrightarrow[s]{rl}\ \{v, e'\}}\quad \text{JO\_RED\_PAIR\_2}$$

$$\frac{}{(\mathbf{pair}\ v)\ v'\ \xrightarrow[s]{\tau}\ \{v, v'\}}\quad \text{JO\_RED\_INPAIR}$$

$$\frac{}{proj1\,\{v, v'\}\ \xrightarrow[s]{\tau}\ v}\quad \text{JO\_RED\_PROJ1}$$

$$\frac{}{proj2\,\{v, v'\}\ \xrightarrow[s]{\tau}\ v'}\quad \text{JO\_RED\_PROJ2}$$

$$\frac{e\ \xrightarrow[s]{rl}\ e'}{\mathbf{left}\ e\ \xrightarrow[s]{rl}\ \mathbf{left}\ e'}\quad \text{JO\_RED\_EVALINL}$$

$$\frac{e\ \xrightarrow[s]{rl}\ e'}{\mathbf{right}\ e\ \xrightarrow[s]{rl}\ \mathbf{right}\ e'}\quad \text{JO\_RED\_EVALINR}$$

$$\frac{}{Case\,(\mathbf{left}\ v)\,of\ \mathbf{left}\ x \Rightarrow e|\ \mathbf{right}\ x' \Rightarrow e'\ \xrightarrow[s]{\tau}\ \{v/x\}e}\quad \text{JO\_RED\_EVALCASEINL}$$

$$\frac{}{Case\,(\mathbf{right}\ v)\,of\ \mathbf{left}\ x \Rightarrow e|\ \mathbf{right}\ x' \Rightarrow e'\ \xrightarrow[s]{\tau}\ \{v/x'\}e'}\quad \text{JO\_RED\_EVALCASEINR}$$

$$\frac{e\ \xrightarrow[s]{rl}\ e'}{Case\ e\ of\ \mathbf{left}\ x \Rightarrow e''|\ \mathbf{right}\ x' \Rightarrow e'''\ \xrightarrow[s]{rl}\ Case\ e'\ of\ \mathbf{left}\ x \Rightarrow e''|\ \mathbf{right}\ x' \Rightarrow e'''}\quad \text{JO\_RED\_EVALCASE}$$

```
Definition rules:        47 good    0 bad
Definition rule clauses: 78 good    0 bad
```

A.2 blah

Appendix B

Project Proposal

B.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[29] 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.

B.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.

B.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[23].

As large amount of research has gone into both monadic concurrency and implementations in OCaml, the project will draw inspiration from Claessen[9], Deleuze[11] and Vouillon[31].

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.

B.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.

B.5 Timetable

The project will be split into two week packages

B.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.

B.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 [24, 5], but with a concurrency focus.

B.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.

B.5.4 Week 7 and 8

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

B.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.

B.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.

B.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.

B.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.

B.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.

B.5.10 Week 19 and 20

Finalising the dissertation and incorporating all feedback and polishing.

B.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.