Tamás Kispéter

Monadic Concurrency in OCaml

Part II in Computer Science

Churchill College

May 7, 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 May 7, 2014

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Overview of concurrency . . . 1
- 1.3 Current implementations of a concurrency framework in OCaml . 2
- 1.4 Semantics of concurrency . . . 4
- 1.5 Semantics to logic . . . 5
- 1.6 Logic to runnable code . . . 5

2 Preparation **6**
- 2.1 Design of concurrent semantics . . . 6
- 2.2 Choice of implementation style . . . 7
- 2.3 Design of monadic semantics . . . 7
- 2.4 Tools . . . 9
  - 2.4.1 Ott . . . 10
  - 2.4.2 Coq . . . 12
  - 2.4.3 OCaml . . . 16
- 2.5 Software engineering approach . . . 18
  - 2.5.1 Requirements . . . 18
  - 2.5.2 Development process . . . 19
  - 2.5.3 Back-up . . . 19
  - 2.5.4 Testing plan . . . 20

3 Implementation **21**
- 3.1 The semantics . . . 21
  - 3.1.1 Arrow types . . . 24
  - 3.1.2 Sum types . . . 25
  - 3.1.3 Product types . . . 25
  - 3.1.4 Fixpoint combinator . . . 25
  - 3.1.5 Monadic primitives . . . 28
  - 3.1.6 Fork . . . 28

3.1.7 Computation place holders . . . 30
3.2 Proof assisstant system . . . 33
3.2.1 Outline of the proof assistant code . . . 33
3.2.2 Extractable reduction relation . . . 37
3.3 OCaml system . . . 39
3.3.1 Outline of the OCaml code . . . 40
3.3.2 Hand modifications and justifications . . . 40
3.3.3 Syntactic sugar . . . 42

4 Evaluation **45**
4.1 Theoretical evaluation . . . 45
4.1.1 Methods . . . 45
4.1.2 Properties . . . 45
4.2 Performance . . . 46
4.2.1 Method . . . 47
4.2.2 Examples . . . 47

5 Conclusion **55**

Bibliography **57**

A Full semantics **61**
A.1 blah . . . 68

B Project Proposal **69**
B.1 Introduction of work to be undertaken . . . 69
B.2 Description of starting point . . . 69
B.3 Substance and structure . . . 70
B.4 Criteria . . . 70
B.5 Timetable . . . 71
B.5.1 Week 1 and 2 . . . 71
B.5.2 Week 3 and 4 . . . 71
B.5.3 Week 5 and 6 . . . 71
B.5.4 Week 7 and 8 . . . 71
B.5.5 Week 9 and 10 . . . 72
B.5.6 Week 11 and 12 . . . 72
B.5.7 Week 13 and 14 . . . 72
B.5.8 Week 15 and 16 . . . 72
B.5.9 Week 17 and 18 . . . 72
B.5.10 Week 19 and 20 . . . 72

B.6 Resource Declaration 72

List of Figures

2.1 High level view of the tool chain 10

3.1 Detailed outline of the implementation 22
3.2 Grammar dependencies 23
3.3 Syntax and semantics of arrow types 24
3.4 Syntax and semantics of sum types 26
3.5 Syntax and semantics of product types 27
3.6 Syntax and semantics of the fixpoint operator 28
3.7 Syntax and semantics of the monadic primitives, ret and bind . . 31
3.8 Syntax and semantics of the fork operator 32
3.9 Dependencies in the Coq semantics 33
3.10 Dependencies in the OCaml code 39

4.1 `kpn` process outline 48
4.2 `kpn` execution time 49
4.3 `sieve` process outline 50
4.4 `sieve` execution time 51
4.5 Sieve fork tree 52
4.6 `sorter` process outline 52
4.7 `sorter` execution time 53
4.8 `sorter` memory use 53

Listings

1.1 LWT example . . . 3
1.2 Async example . . . 4
2.1 Ott metavariable definition . . . 10
2.2 Ott grammar example . . . 10
2.3 Ott value subgrammar example . . . 11
2.4 Ott substitution example . . . 11
2.5 Ott reduction relation example . . . 12
2.6 Ott single reduction . . . 12
2.7 Coq Prop logic example . . . 13
2.8 Coq Prop predicate example . . . 13
2.9 Coq Prop new predicate example . . . 13
2.10 Coq inductive data structure example . . . 13
2.11 Coq coinductive data structure example . . . 13
2.12 Coq fixpoint example . . . 14
2.13 Coq theorem example . . . 14
2.14 Coq to OCaml extraction of seq . . . 15
2.15 Coq to OCaml extraction of length . . . 15
2.16 Coq logical inductive example . . . 15
2.17 Coq to OCaml extraction of a logical inductive relation . . . 16
2.18 OCaml simple function example: square . . . 16
2.19 OCaml recursive function example: factorial . . . 16
2.20 OCaml complex function example: insertion sort . . . 17
2.21 OCaml imperative function example . . . 17
2.22 OCaml evaluation function example . . . 18
3.1 Ott type expressions . . . 33
3.2 Coq type expr . . . 34
3.3 Ott value subgrammar . . . 34
3.4 Coq value subgrammar . . . 35
3.5 Coq expression substitution . . . 36
3.6 Ott reduction relation example . . . 36

3.7 Coq reduction relation example 37
3.8 Coq label equality lemma 37
3.9 Coq co-inductive selection sequence 38
3.10 Coq reduction clause with unsafe assumption 39
3.11 Coq extractable reduction clause with safe assumption 39
3.12 OCaml lazy selectstar 40
3.13 OCaml stream selectstar 40
3.14 OCaml original substitution case 41
3.15 OCaml original application case 41
3.16 OCaml fixed substitution case 41
3.17 OCaml swapped assumptions 42
3.18 OCaml round-robin scheduler 43
3.19 OCaml random scheduler 43

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 more and more subtle bugs that even extensive unit testing could not find are discovered. 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[28], C++ and Java[24], 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[32]. 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[34] 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

Is Listing 1.1 an example 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 I defined `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[25] implements join calculus over OCaml, Functory[16] focuses on distributed computation, OCamlNet exploits multiple cores and OCamlMPI[20] 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 full[12, 19] and delimited[17] continuations[30], trampolined style[13], continuation monads[9], promise monads[23] and event based programming (as used, for example in the OCamlNet[31] 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, 21]. Coq[5], HOL and Isabelle are widely used proof assistants. Tools like Ott[29] 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[22] 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

The preparation phase of this project involved many decisions, 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, and sometimes also a unit of resources. This algebra typically comes with a number of operations. For example,

- $P \mid Q$ 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. I have left out formal treatment of communication channels in order to limit the scope of the project.

2.2 Choice of implementation style

Deleuze[11] surveys a number of implementation styles of lightweight concurrency for OCaml. The styles fall in two broad categories: direct and indirect styles. The direct style involves 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. Within this style there are various approaches to using continuations. Two examples that have been explored in OCaml are full or call/cc and delimited continuations.

A full continuation captures the entire current control stack and passes it on as a first class value. When this value is "applied" the system replaces the control stack with the captured stack. Leroy explores call/cc style semantics[19] for OCaml, however advises against the use of his library.

Delimited continuations set boundaries on this capture process and only capture part of the control stack. Kiselyov developed an OCaml library for delimited continuations[17, 18] and advocates it over call/cc style.

Indirect styles include the trampolined style and two monadic styles: continuations and promises.

Trampolined style, as Ganz[13] describes, involves a scheduler and a way to box up computation into either `Doing of unit -> T 'a` for unfinished computations or `Done of 'a` for values. The scheduler then can pick which one of these boxes to evaluate.

Monadic styles also box up computations: continuation monads box up continuations[9], while promise monads[23] box up value place holders that would be the finished product of a computation. Both of these monadic approaches have a further requirements in the way operations relate to each other. LWT is designed with continuation monads in mind, while Async is based on the promise monad.

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

A monad is a way to package up a computation and operations to combine and manipulate packaged computations. Monads are a popular approach to organ-

ising code that may have a side-effect. Input/output, concurrency, exceptions and foreign language interfaces are applications of this concept. Most presentations of monads 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 Input/output monad would often have IO as the marker. Furthermore, there are two operations: ret and bind.

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

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 $\alpha \rightarrow \mathbf{con}\,\beta$). Bind then returns a $\mathbf{con}\,\beta$. With types $\gg=$ means: $\gg= : \forall\alpha\,\beta.\ \mathbf{con}\,\alpha \rightarrow (\alpha \rightarrow \mathbf{con}\,\beta) \rightarrow \mathbf{con}\,\beta$.

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

1. ret is a left neutral element of bind:

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

2. ret is a right neutral element of bind:

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

3. bind is associative:

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

I will return to the exact nature of the equivalence relation $\equiv$ used in this project in the evaluation section.

The concept of a monad comes from category theory. Category theory is a general tool that is often used to model functional programming languages and programs. The monad concept in category theory has a slightly different, but equivalent representation. Instead of bind and ret, there are three operations: (T, η, μ), commonly called lift or unit, map and join. Lift packages an object, map takes a function between regular objects and returns a function between the packaged counterparts of the objects and join unpackages a layer on a doubly packaged object. These three operations have to satisfy two conditions, called coherence conditions.

1.

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

Or as commutative diagram:

$$\begin{array}{ccc} T^3 & \xrightarrow{T\mu} & T^2 \\ {\scriptstyle \mu T}\downarrow & & \downarrow{\scriptstyle \mu} \\ T^2 & \xrightarrow[\mu]{} & T \end{array}$$

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

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

   Or as commutative diagram:

$$\begin{array}{ccc} T & \xrightarrow{\eta T} & T^2 \\ {\scriptstyle T\eta}\downarrow & \searrow & \downarrow{\scriptstyle \mu} \\ T^2 & \xrightarrow[\mu]{} & T \end{array}$$

   That is to say packaging and then subsequently unpackaging an object behaves as an identity.

While this formulation is equivalent to the parametric type, bind and ret approach it is rarely used mainstream programming languages.

2.4 Tools

The project uses a chain of three tools:

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

I spent the preparation phase acquainting myself with all three of these systems, as I have not used them before for any serious work.

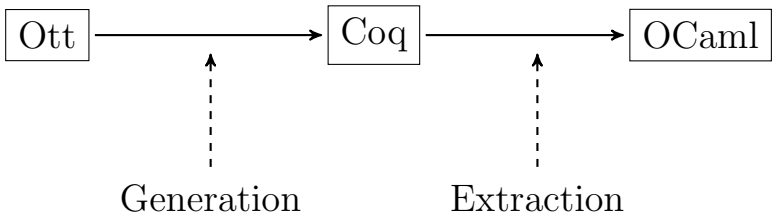


Figure 2.1: High level view of the tool chain

2.4.1 Ott

To avoid duplication of the semantics in several formats I have to chosen to use a supporting tool called Ott[29]. 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. Ott is the primary form of the semantics from which all further forms are derived 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, such as 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.

```
1 v :: 'v_' ::=                                    {{ com value    }}
2   | \ x . t       ::   :: Lam                    {{ com lambda   }}
3
4 subrules
5   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.

```
1 substitutions
2   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 premises appear line-by-line above the ascii-art line, and the result below the line. Next to the line is the name of the statement, prefixed by the name of the relation to avoid ambiguity.

2.4.2 Coq

Ott is able to generate output for a number of proof assistants, including Coq and Isabelle, which both 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 a formal proof assistant with a mathematical higher-level language called *Gallina*, based around the Calculus of Inductive Constructions. Gallina

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 are divided into three sorts: Prop (propositions), Type (types) and Set (sets). 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

Coq can define predicates and relations over sets. In Listing 2.8 I have defined a proposition, a tautology that if the product of two integers is zero, then at least one of them must be zero.

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

Listing 2.8: Coq Prop predicate example

New predicates can be defined inductively. Listing 2.9 defines the mutually inductive predicates odd and even.

```
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 has 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, called tactics, 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. Logical inductive systems are inductively defined propositions and fixpoints involving propositions. The addition relation in Listing 2.16 of three elements is one such logical inductive construct.

```
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

Logical inductive types do not need to conform to any input/output relationship and computations that correspond to a logical inductive relation need not always terminate. Extraction from Coq is required to produce a certification of equivalence and the Calculus of Inductive Constructions, the logic underlying Coq, cannot directly express a certification for a non-terminating program. .

However with the help of a plugin developed by David Delahaye et al [10, 33] by marking different modalities of the inductively generated proposition I

can generate code with an input-output convention. In the case of the addition relation, by marking the first two parameters as inputs and the third as output, the plugin can extract a functioning recursive construct as show in Listing 2.17.

```
(** 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, supports 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.

```
type new_int_list =
    | Empty
    | Cons of int * new_int_list

let rec new_iter f l =
  match l with
  | Empty -> ()
  | x :: t -> f x; new_iter f t

let rec sigma f = fun l ->
    let res = ref 0 in
    let add_f x = res := (f x) in
    new_iter add_f l; !res
```

Listing 2.21: OCaml imperative function 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 2.22: OCaml evaluation function example

2.5 Software engineering approach

2.5.1 Requirements

As I described earlier and in the Project Proposal, the project must satisfy the following criteria:

- The basis of the project must be a clear semantic description.
- This description must have a faithful proof assistant counterpart.
- There must be a runnable OCaml version that is faithful to the above.
- This runnable OCaml code must be tested against existing implementations of lightweight concurrency.

The project should also include the following features:

- The extracted code and the semantics should be proven to be the same.
- Monadic laws should be obeyed by the implementation.
- The concurrency behaviour should exhibit properties required by process calculi.

Further extensions that the project could have:

- Like most theoretical languages, a proof of type preservation and progress would be good feature.
- A typechecker would be beneficial for potential users.
- Extensive syntactic sugar could greatly improve the use of the framework.
- A hand optimized version of the extracted semantics might prove viable as compared to other implementations.

After Part II I want to extend the project with the following features:

- A formal treatment communication between threads. Channel semantics was not within the scope of this project
- A documentation on how to use the project: both from a user's perspective and from a developer/researcher's perspective.

2.5.2 Development process

I used a spiral development pattern in the project. This pattern seemed well suited for fast paced development that is required by the Part II schedule. However, at a later stage it became apparent that a waterfall model might have been better suited for this project: with the full implementation chain, shown in Figure 3.1, the project had high viscosity. Each change in the initial semantics required between a few hundred to thousands of lines of change throughout the system.

2.5.3 Back-up

Throughout the project I have made frequent back-ups in the form of a Dropbox account, a Google Drive account and an infrequently used back-up drive. At the initial stages the multiple cloud storage copies were invaluable as I had trouble setting up a working environment on my personal laptop. With the seamless synchronisation I could work on an MCS machine and immediately switch back to my personal computer when it was possible. I used git on GitHub as a version control system. Transparency of development was important for me, therefore my overseers, my supervisor and my Director of Studies all had access to the version control system and the automatically updated Dropbox account.

2.5.4 Testing plan

I identified a set of theoretical properties to evaluate and if possible prove. Where possible, I preferred formal proof over unit or behavioural testing as formal proof can show the absence of bugs, not just the presence of them.

I only discovered the performance evaluation framework used in Section 4.2 at a very late stage, however it greatly increased the amount of data that was collected.

Chapter 3

Implementation

This concurrency framework was implemented as a small language of expressions and a reduction relation. These two were then translated to OCaml as constructors and an evaluation function respectively. To use the framework the user constructs an expression of the language, including within the expression the computations he wants to evaluate. The expression is than passed the evaluation function. This language provides tools for sequencing, running computations in parallel and to make decisions based on what computations have finished. The implementation of the language was done in 6 stages as shown in Figure 3.1.

The overall semantics were defined in Ott. These semantics were then extracted to Coq by Ott in a logical inductive format. This format, as previously mentioned, is well suited for proofs, but not for extraction. Furthermore, Ott did not correctly handle values that are partial applications of primitives taking more than one argument (**fork** and **pair**). The incorrect value predicate is fixed in the Modified Coq stage appearing in Figure 3.1. This is the only change.

I modified the logical inductive format of the semantics in the Extractable Coq stage to have a well formed input for the extraction plugin. This plugin will be detailed later. After the extraction the generated OCaml contains computation place holders and unused labels for the transitions. I replace these place holders with the actual computations and provide some syntactic sugar, as the generated OCaml is rather cumbersome to write in.

3.1 The semantics

In this section I will describe the implemented semantics. These semantics were written in Ott. Much of the inspiration for the semantics of basic features comes from Benjamin C. Pierce: *Types and Programming Languages*[27]. The Ott code

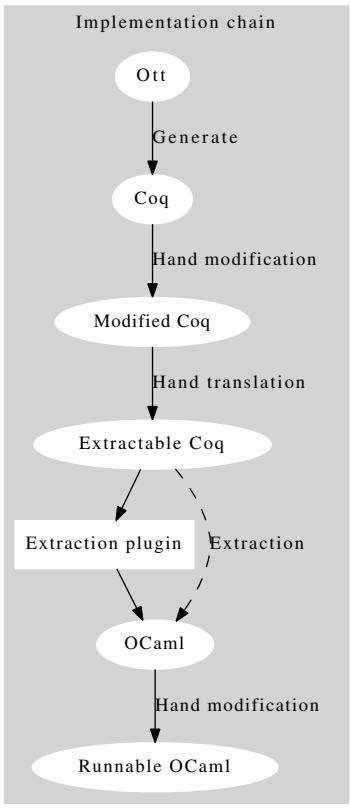


Figure 3.1: Detailed outline of the implementation

was based on the simply typed λ-calculus and polymorphic λ-calculus examples provided with the tool[29]. For a LaTeX render of the full semantics as produced by Ott see appendix A.

The language of this concurrency framework is simply typed.

The reduction semantics is given as small step labelled transitions. Small step semantics means that there is a reduction relation that is between two expressions e, e' where e can directly and atomically transition to e'. Labelled transitions further extend this idea. In this project I used a 4-tuple (e, s, rl, e') of the starting expression, a selection operator that will be supplied by a scheduler to pick between potential reductions, a reduction label that describes the observable action and the ending expression of the transition. What this four tuple means that given the selection operator, which may be 1 or 2, given e as a starting expression can move to e' and with an effect rl. This rl may be an atomic action l that can be observed or be τ which is a silent action. Silent actions are not observed from the outside. I used the notation $e \xrightarrow[s]{rl} e'$.

The semantics splits into three main parts

1. Grammars:
   - expressions
   - values
   - constants
   - types
   - terminals

   For the dependence between different parts of the grammars, see Figure 3.2.
2. Type judgements
3. Reduction judgements

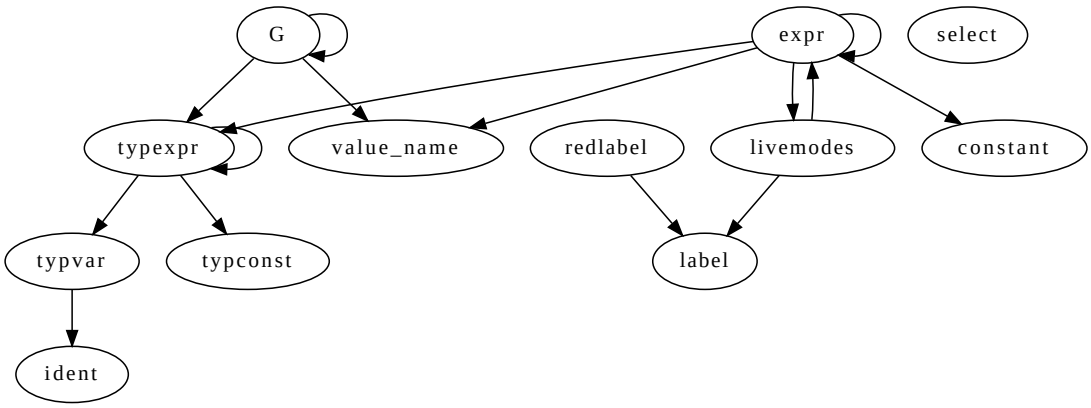


Figure 3.2: Grammar dependencies

I will detail the semantics feature-by-feature: arrow types or functions, sum types or tagged unions, product types or pairs, the fixpoint combinator, the monadic primitives and the fork operator. The presentation of semantics for each of the features were inspired by Pierce[27]. Each section will have a corresponding table of syntax, evaluation rules and typing rules. In the syntax section I will introduce every new syntactic form used in the evaluation and typing rules, but I will not repeat previously mentioned syntax. The presentation of both the syntax and the evaluation and typing rules were slightly simplified for better readability.

Syntax

e ::=		*terms:*
	x	*value name*
	λx:T.e	*abstraction*
	e e	*application*
v ::=		*values:*
	λx:T.e	*abstraction value*
T ::=		*types:*
	TV	*type variable*
	T→T	*→ type*
Γ ::=		*contexts:*
	∅	*empty context*
	Γ, x:T	*term variable binding*
rl ::=		*labels:*
	τ	*silent*
	l	*action*
s ::=		*select:*
	1	*first*
	2	*second*

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

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

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

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

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

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

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

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

Figure 3.3: Syntax and semantics of arrow types

3.1.1 Arrow types

Arrow types or functions are ubiquitous in functional programming languages. In Figure 3.3 I detail the basic syntax and semantics of functions. The style and details are based on Pierce[27, p. 103]. A function abstraction $\lambda x : T.e$ encloses a yet unreduced expression e that may involve the variable x that is bound by the abstraction. I used call-by-value semantics for functions and head-first reduction. Furthermore, arguments are explicitly annotated in functions and

substitution does not provide facilities for renaming. It is presumed that the user of the framework takes care of picking fresh variable names.

3.1.2 Sum types

Many circumstances require the ability to describe expressions that are either one type or the other. Sum types or otherwise known as labelled unions are a simple solution to this. An expression e that has type T can be labelled as a variant in a sum type by attaching a label **left** e and **right** e will give force it to have type $T + T'$ and $T' + T$ for some type T'. This expression can be then destructed by the expression "Case e of **left** $x_1 \Rightarrow e_2$ | **right** $x_2 \Rightarrow e_3$ " which will evaluate to either a substitution of e_2 or e_3 depending on what the tag said. In this project I use sum types in the signature of **fork** that will be detailed later. In Figure 3.4 I detail the implementation of sum types that is based on Pierce[27, p. 132].

3.1.3 Product types

One very common feature of functional languages is pairs or product types. Grouping different types of data together in one logical unit often makes code more simple. For example computations with complex numbers would be rather unintuitive if we always had to handle the real and imaginary parts separately. To define products, first I introduce the syntax $\{e, e'\}$, the pair of expressions e and e'. If they had types T and T' respectively I define the type construct for this $T \star T'$. I have introduce three primitive functions to deal with pairs: **pair** places two values in a pair, **proj1** takes the first element of a pair and **proj2** takes the second.

In Figure 3.5 I describe the precise implementation of product types defined in this language. The style and details are based on Pierce[27, p. 126].

3.1.4 Fixpoint combinator

Recursion is very characteristic of functional programming languages. There are many ways to achieve recursive constructs in terms and even in types. A very elegant treatment surfaced from the formal study of recursive constructs in the form of fixpoint combinators. In denotational semantics for example loops and other recursions are treated as the greatest fixpoints of continuous functions. A fixpoint combinator y is defined as a term that given function f satisfies

$$y\, f \;\equiv\; f\,(y\, f)$$

Syntax

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

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

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

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

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

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

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

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

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

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

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

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

This axiom poses a requirement on the type of y, it should be $(T \to T) \to T$ In untyped λ-calculus there are many simple terms that can achieve this, for example the Y combinator:

$$Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

However in languages like the one implemented in this chapter, where arguments are always reduced before the function application can begin the Y combinator would always diverge. There are more than one option for implementation in

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

languages like this, but I have chosen to use the one described in Pierce[27, p. 144]. This style fits well with the tool chain I used as it does not require another partial function application and it is also very minimal.

3.1.5 Monadic primitives

The concurrency monad consists of a parametric type **con** T, where T is a type parameter describing the type of computation or value enclosed and three key operations

- ret, also known as return. It has type $T \to$ **con** T and simply evaluates its parameter and boxes up the result in the parametric type
- $\gg=$, also known as bind, sequences two operations. The second argument is the continuation for the first parameter. More formally it has a type **con** $T_1 \to (T_1 \to$ **con** $T_2) \to$ **con** T_2, 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.

3.1.6 Fork

Up until this point I have not mentioned any language feature that implements concurrency which is the main focus of the dissertation. The third, additional operation of the concurrency monad is fork, which is the way to spawn new threads (two in this particular case). There are a number of ways to implement

fork and indeed this project went through various iterations of semantics for this operation.

As a first approximation I had to decide on a signature for fork. On the argument side fork may take zero, one, two or many arguments. No arguments would be reminiscent of the UNIX system call `fork()` where the two paths are distinguished by replacing the fork call with differing values. This approach would result in copying potentially large expressions and a more complex evaluation context, that is the terms used for the source and destination in the reduction relation. For example, Jones[14] chose an implementation with one argument, however with a similar requirement of a metalanguage of parallel terms $e \mid e'$. I chose to go with two arguments as that can maintain a simple evaluation context with reductions from language expression to expression. The choice between these three argument styles is largely arbitrary as they all implicitly form the binary parallel composition $e \mid e'$. Note however that several arguments can be elegantly simulated by composing binary parallel compositions.

To stay within the monad I have chosen to have the arguments as already boxed terms, that is of type $\mathbf{con}\ T_1$ and $\mathbf{con}\ T_2$ for some T_1, T_2. For better expressibility, I chose to take the two argument curried, that is **fork** is a higher-order function: given the first argument it returns a function that takes only one parameter and will then behave as the **fork**.

$$\mathbf{fork} : \mathbf{con}\ T_1 \to (\mathbf{con}\ T_2 \to R)$$

where R is the yet not described return type. Currying allows partial application of **fork** and to be passed around as a value with only one edge filled. The other option would have been to take a pair of values, however that would have limited the variety of constructions.

To keep within the monad, I required that R be a concurrent type, that is $R = \mathbf{con}\ R'$ for some R' type. There are many choices available for the return type. Other popular concurrency primitives have varying return semantics. For example **join** would return the pair of values resulted from the two expressions giving $R = \mathbf{con}\ (T_1 \star T_2)$. Another primitive, **choose** would pick one and discard the other: $R = \mathbf{con}\ (T_1 + T_2)$. I wanted to provide a combination of this: to signal which thread has finished first, but keep the partially reduced other edge around, so the user can use it. From the previous constructions a return type as $R = \mathbf{con}\ ((T_1 \star \mathbf{con}\ T_2) + ((\mathbf{con}\ T_1) \star T_2))$ would implement this behaviour. At first glance, this seems to be a rather complex signature but it is versatile enough to implement both other primitives and capture the semantics of a wide range of problems well. The return type also raises the question of interleaving

semantics: as I am implementing concurrency in software I was not constrained by hardware to interleave reductions. It would be perfectly acceptable to reduce both edges of a fork at the same time if possible. Indeed this project was originally designed with not necessarily interleaving semantics in mind. However, that lead to a blow up in the number of rules, the complexity of signature ($R = \mathbf{con}\ ((T_1 \star \mathbf{con}\ T_2) + ((\mathbf{con}\ T_1) \star T_2) + (T_1 \star T_2))$) and the number of potential behaviours of the system. I chose to simplify to interleaving semantics for the theoretical simplicity over potential efficiency gains.

Putting this all together gives the full signature of **fork** as

$$\mathbf{fork} : \mathbf{con}\ T_1 \rightarrow (\mathbf{con}\ T_2 \rightarrow \mathbf{con}\ ((T_1 \star \mathbf{con}\ T_2) + ((\mathbf{con}\ T_1) \star T_2)))$$

In Figure 3.8 I give the detailed semantics of fork. There is no need to describe reduction rules for arguments not of the form **fork** $(\mathit{Live}\ lm)\,(\mathit{Live}\ lm')$ as the application rule in Figure 3.3 has already reduced terms to values and due to the typing relation of **con** types (as defined in Figure 3.7) only allows values tagged with *Live*. Notice that this is the first time the selection operator of the reduction relation is explicitly specified. A selection value of 1 will reduce the first edge of the fork, a value of 2 will reduce the second edge. When an expression value of the for *Live expr v* is encountered it reduces to the respective tagged value. It is important to note that when a computation is encountered it "runs" that computation and immediately finishes the **fork**. The "result" of such computation is a unit value for simplicity.

3.1.7 Computation place holders

In previous subsections I have referenced computation place holders of the form *Live comp l*. These are the holes that will be filled by actual OCaml **unit** → **unit** functions to be evaluated. The reductions in Figures 3.7 and 3.8 that evaluate such place holders are modified in the runnable OCaml code to just run the functions within these holes. The purpose of the labels is purely logical: they serve as the tool to prove that the sequence of actual code calls obey certain properties. These labels are not necessary for the runnable OCaml code.

The decision that these always reduce to unit was one purely of scope: further work on this project could include reducing these expressions within the framework or other forms of dynamic information.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Syntax

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

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

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

Figure 3.8: Syntax and semantics of the fork operator

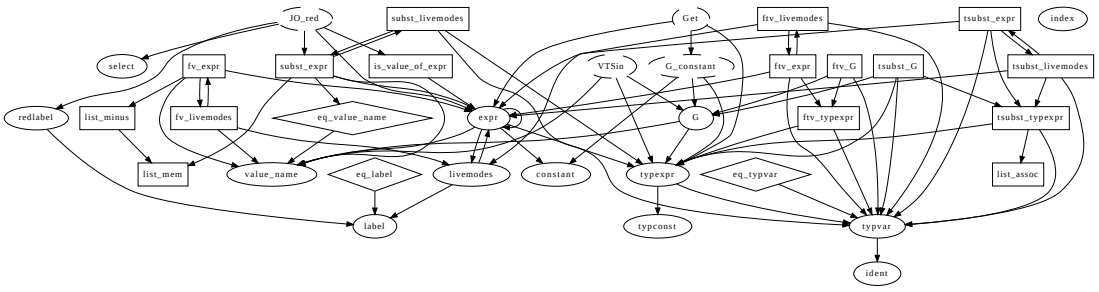


Figure 3.9: Dependencies in the Coq semantics

3.2 Proof assisstant system

In this section I briefly outline the structure of the translated proof assistant representation and how I modified it to be extractable to OCaml.

3.2.1 Outline of the proof assistant code

In Figure 3.9 I detail the dependencies between different parts of the Coq version of the semantics. There are four types of objects in the Coq file:

1. Inductive sets: they are the inductive data structures. These are the expression grammar objects mentioned in Figure 3.2 with the same structure and the types of metavariables. Inductive sets are denoted by ovals with solid outlines. As an example, type expressions are translated from the Ott version in Listing 3.1 to the Coq equivalent in Listing 3.2. Simple grammars generate a set of tagged variants.

```
typexpr, t :: TE_ ::=
  | typconst                          ::    :: constants
  | typvar                            ::    :: var
  | typexpr -> typexpr'               ::    :: arrow
  | typexpr '*' typexpr'              ::    :: prod
  | con typexpr                       ::    :: concurrent
  | typexpr '+' typexpr'              ::    :: sum
  | ( typexpr )                       :: S :: paren {{ ich [[
    typexpr]] }} {{ ocaml [[typexpr]] }}
```

Listing 3.1: Ott type expressions

```
Inductive typexpr : Set :=
 | TE_constants (typconst5:typconst)
 | TE_var (typvar5:typvar)
 | TE_arrow (typexpr5:typexpr) (typexpr':typexpr)
 | TE_prod (typexpr5:typexpr) (typexpr':typexpr)
 | TE_concurrent (typexpr5:typexpr)
 | TE_sum (typexpr5:typexpr) (typexpr':typexpr).
```

Listing 3.2: Coq type expr

2. Fixpoints, the functions in Coq: The functions in this are all automatically generated from the Ott file. The most important ones are the expression substitution, free variable, value check fixpoints. The script also includes a few functions supporting the previous ones. Fixpoints are denoted by rectangles. The value subgrammar is translated as a fixpoint over expressions.

```
value, v :: V_ ::=
  | constant                                  ::   :: constant
  | function value_name : typexpr -> expr       ::   ::
   function
  | live lm                                 ::   :: live_expr
  | inl v                                     ::   :: taggedleft
  | inr v                                     ::   :: taggedright
  | { v , v' }                                ::   :: valuepair
  | ( v )                                   :: S :: paren {{ ich [[v
   ]] }} {{ ocaml [[v]] }}
```

Listing 3.3: Ott value subgrammar

```
Fixpoint is_value_of_expr (e_5:expr) : Prop :=
  match e_5 with
   | (E_ident value_name5) ⇒ False
   | (E_constant constant5) ⇒ (True)
(* Manual change *)
   | (E_apply expr5 expr') ⇒ match expr5 with | E_constant (
    CONST_fork) ⇒ is_value_of_expr (expr') | E_constant (
    CONST_pair) ⇒ is_value_of_expr (expr') | _ ⇒ False end
   | (E_bind expr5 expr') ⇒ False
   | (E_function value_name5 typexpr5 expr5) ⇒ (True)
   | (E_fix e) ⇒ False
   | (E_live_expr lm) ⇒ (True)
   | (E_pair e e') ⇒ ((is_value_of_expr e) ∧ (is_value_of_expr e
    '))
   | (E_taggingleft e) ⇒ ((is_value_of_expr e))
   | (E_taggingright e) ⇒ ((is_value_of_expr e))
   | (E_case e1 x1 e2 x2 e3) ⇒ False
end.
```

Listing 3.4: Coq value subgrammar

Notice however, that I was not able to define the value property of partial applications of primitives **fork** and **pair**, therefore I manually changed the incorrect value subgrammar check. Ott offers single and multiple substitution predicates for its destination languages. These are implemented as fixpoints in Coq. As an example, see the expression substitution in Listing 3.5.

```
Fixpoint subst_expr (e_5:expr) (x_5:value_name) (e__6:expr) {
    struct e__6} : expr :=
  match e__6 with
   | (E_ident value_name5) ⇒ (if eq_value_name value_name5 x_5
    then e_5 else (E_ident value_name5))
   | (E_constant constant5) ⇒ E_constant constant5
   | (E_apply expr5 expr') ⇒ E_apply (subst_expr e_5 x_5 expr5)
    (subst_expr e_5 x_5 expr')
   | (E_bind expr5 expr') ⇒ E_bind (subst_expr e_5 x_5 expr5) (
    subst_expr e_5 x_5 expr')
   | (E_function value_name5 typexpr5 expr5) ⇒ E_function
    value_name5 typexpr5 (if list_mem eq_value_name x_5 (cons
    value_name5 nil) then expr5 else (subst_expr e_5 x_5 expr5))
   | (E_fix e) ⇒ E_fix (subst_expr e_5 x_5 e)
   | (E_live_expr lm) ⇒ E_live_expr (subst_livemodes e_5 x_5 lm)
   | (E_pair e e') ⇒ E_pair (subst_expr e_5 x_5 e) (subst_expr
    e_5 x_5 e')
   | (E_taggingleft e) ⇒ E_taggingleft (subst_expr e_5 x_5 e)
   | (E_taggingright e) ⇒ E_taggingright (subst_expr e_5 x_5 e)
   | (E_case e1 x1 e2 x2 e3) ⇒ E_case (subst_expr e_5 x_5 e1) x1
     (subst_expr e_5 x_5 e2) x2 (subst_expr e_5 x_5 e3)
end
```

Listing 3.5: Coq expression substitution

3. Logical inductive sets: As mentioned previously a logical inductive set is an inductively defined set of propositions. This is the way the reduction relation and the typing relation is represented. Logical inductive sets are represented in Figure 3.9 by ovals with dashed outline. For example a clause in the reduction relation is translated from the simple inference rule presentation in Listing 3.6 to proposition in the logical inductive set `JO_red` in Listing 3.7.

```
e [ s ] —> [ rl ] e''
————————————————————————  :: context_app1
e e' [ s ] —> [ rl ] e'' e'
```

Listing 3.6: Ott reduction relation example

```
1 Inductive JO_red : expr → select → redlabel → expr → Prop :=
      (* defn red *)
2  | JO_red_context_app1 : forall (e e':expr) (s:select) (rl:
     redlabel) (e'':expr),
3       JO_red e s rl e'' →
4       JO_red (E_apply e e') s rl (E_apply e'' e')
5   ...
```

Listing 3.7: Coq reduction relation example

4. Lemmas: There are a few supporting lemmas about the equality of variables. These are automatically generated by Ott. Lemmas are denoted with diamonds. An simple example would be the straightforward lemma that says labels are either equal or not in Listing 3.8.

```
1 Lemma eq_label: forall (x y : label), {x = y} + {x <> y}.
2 Proof.
3   decide equality; auto with ott_coq_equality arith.
4 Defined.
```

Listing 3.8: Coq label equality lemma

The automatically generated Coq representation had a slight issue in that I could not easily define partial application of curried primitive functions as values. Instead, I hand modified the fixpoint in the translated Coq the particular case on Line 6 in Listing 3.4.

3.2.2 Extractable reduction relation

As I mentioned in Section 2.4.2, Coq provides extraction facilities to OCaml and Haskell, but not from inductive sets and fixpoints that involve propositions, more specifically the *Prop* sort.

Logical inductive types are often used for description of various concepts in semantics, especially reduction relations. Ott generates a logical inductive relation as seen in Listing 3.7. For simple relations, like the value relation in Listing 3.4 it is simple to rewrite in a set inductive manner. However, for complex non-terminating relations, like the reduction relation in this project, rewriting is not feasible. Delahaye et al.[10, 33] proposed a method of extracting such relations to OCaml by annotating the input/output modes of the elements of the relation.

I rewrote the value check logical fixpoint to an extractable version by simply replacing the True and False propositions by their boolean counterpart and successfully extracted it by the built-in Coq extraction. However, not even the Coq

plugin based on element modes could extract the reduction relation generated by Ott.

The first issue with the extraction to a functional program was the way the selection argument was supplied. Originally, the reduction relation only required a single selection value. In the case of fork reduction rules this would not be enough to generate a certified program.

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

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

The extracted program would have to somehow pick a possible value for s in $e \xrightarrow[s]{rl} e'$ while it is only supplied with the value 1. Always picking one or the other would not be satisfactory. As I had no way of choosing it non-deterministically, I wanted the argument to fully describe all s values the system used. I opted for a co-inductive stream of selections because there can be an arbitrary number of **fork**s nested in each other with arbitrary number of selection values to make a single step .

```
CoInductive selectstar : Set := Seq : select → selectstar →
    selectstar.
```

Listing 3.9: Coq co-inductive selection sequence

Co-inductive data structures represent potentially infinite data and featured in a number of programming languages: lazy lists, trees and streams all describe co-inductive structures. In Listing 3.9 I define `selectstar` where each element is pair of a selection value and a further `selectstar` representing the rest of the selection values. With an infinite sequence of selection values the program can make arbitrary many decisions and it is extractable.

The second problem that surfaced was that the extraction plugin does not generate code that backtracks from a case where the reduction relation is invoked on an internal part. If an expression e does not reduce and the extracted function is called it will fail with an assert false. I chose to add a logically superfluous assumption of expression e not being a value. This assumption is superfluous as an expression that reduces cannot be a value by the `red_not_value` theorem. Listings 3.10 and 3.11 are a good example of how this transformation happens.

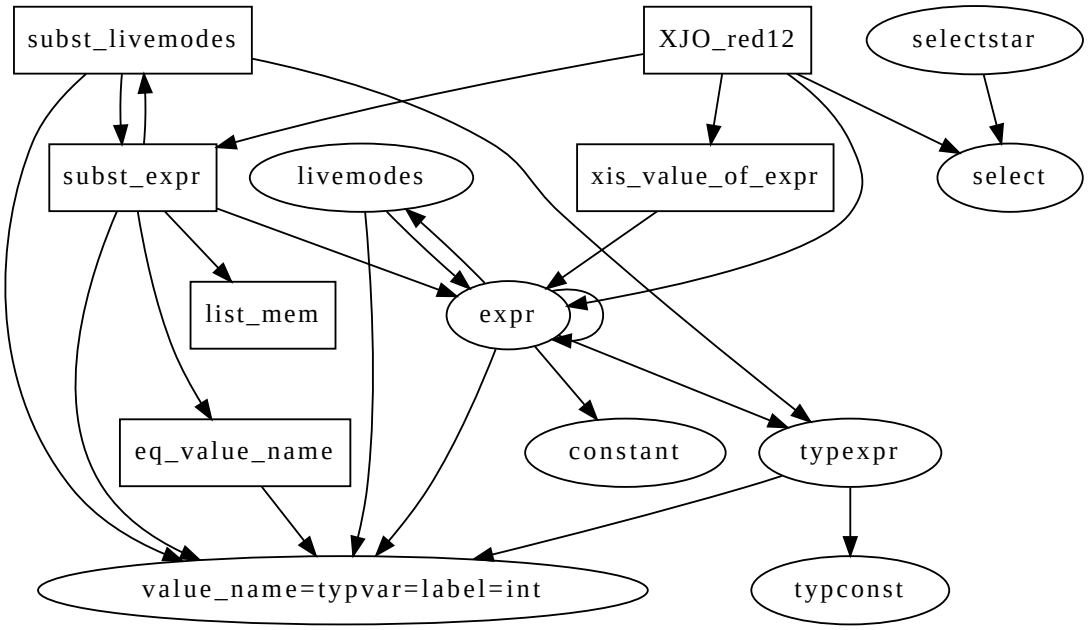


Figure 3.10: Dependencies in the OCaml code

```
| JO_red_evalbind : forall (e e'':expr) (s:select) (rl:
    redlabel) (e':expr),
      JO_red e s rl e' →
      JO_red (E_bind e e'') s rl (E_bind e' e'')
```

Listing 3.10: Coq reduction clause with unsafe assumption

```
| XJO_red_evalbind : forall (e e'' e':expr) (s:selectstar),
    (eq (xis_value_of_expr e) false) →
      XJO_red e s e' →
      XJO_red (E_bind e e'') s (E_bind e' e'')
```

Listing 3.11: Coq extractable reduction clause with safe assumption

The last issue was due to the experimental nature of the plugin: the optimisations and inference algorithm ran out of stack space when invoked with all 26 rules. Therefore I extracted in three parts and recombined them by hand.

3.3 OCaml system

In this section I give a brief outline of the structure of the extracted OCaml, how it is modified to be runnable and a brief overview of potential syntactic sugar that can be used to aid development in the framework.

3.3.1 Outline of the OCaml code

The OCaml code consists of the extracted versions of the following:

- Inductive sets like expressions, constants and type expressions as tagged variants. These are represented as ovals in Figure 3.10.
- Functions like the expression substitution or XJO_red12, the reduction function. These are the rectangles in Figure 3.10.
- The metavariables `value_name, label, ident` are all of OCaml type int instead of constructor based extractions of Coq nat. This has the caveat that int may overflow or represent negative numbers, while the Coq nat cannot. Realistically, no program would have this problem.

Their interdependence is shown on Figure 3.10.

3.3.2 Hand modifications and justifications

I have made a number of modifications to the OCaml code to make it runnable. This includes changing the type of labels from int to **unit** → **unit** and inserting terms in the relevant cases to run these computations.

Furthermore I have changed the way `selectstar` is implemented. The extraction results in the type in Listing 3.12 which involves the Lazy module of OCaml. While the extraction of co-inductive types to lazy types is sound, for simplicity I used a simple lazy stream in Listing 3.13.

```
1 type selectstar = __selectstar Lazy.t
2 and __selectstar =
3 | Seq of select * selectstar
```

Listing 3.12: OCaml lazy selectstar

```
1 type selectstar = | Seq of select * (unit -> selectstar)
```

Listing 3.13: OCaml stream selectstar

As the extraction plugin is experimental there were a number of inefficiencies and a few issues due to the fact that the semantics of OCaml pattern matching is sequential, while it is parallel in Coq. While Tollitte[33] made progress on the merging of cases when one case subsumes the other, these were not always correctly identified. For example the case Lines 7 to 10 in Listing 3.15 subsumes the case Listing 3.14. The plugin was unable to infer this as that would have required it to show that function term is always a value, regardless of its parameters. This

is an issue as any function with a parameter that can be reduced will fail, as the case in Listing 3.14 comes first.

```
1  | (E_apply (E_function (x, t, e), v), s) ->
2    (match xis_value_of_expr v with
3     | true -> subst_expr v x e
4     | _ -> assert false (*  *))
```

Listing 3.14: OCaml original substitution case

```
1  | (E_apply (e, e'), s) ->
2    (match xis_value_of_expr e with
3     | false ->
4       (match xJO_red12 e s with
5        | e'' -> E_apply (e'', e')
6        | _ -> assert false (*  *))
7     | true ->
8       (match xJO_red12 e' s with
9        | e'' -> E_apply (e, e'')
10       | _ -> assert false (*  *))
11    | _ -> assert false (*  *))
```

Listing 3.15: OCaml original application case

My solution was to insert the correct step into the failing match as in Listing 3.16. Note, that taking that reduction may fail, but that is the expected behaviour.

```
1  | (E_apply (E_function (x, t, e), v), s) ->
2    (match xis_value_of_expr v with
3     | true -> subst_expr v x e
4     | false -> E_apply (E_function (x, t, e), (xJO_red12 v s)))
```

Listing 3.16: OCaml fixed substitution case

A reoccurring inefficiency comes from the extraction plugin generating a default failing case in all situations, even if the default case may never happen. Line 11 in Listing 3.15 is an example of this: a boolean may only take the values true or false. A similar issue occurs when evaluating a function: Line 4 in the same listing matches by giving a name to the return value, but generates a default case as well. To this latter problem I simply removed the match and replaced the name of the return value with the function call.

A further issue occurs when occasionally the plugin reorders matches on assumptions. Even though the safe assumption was inserted in the case in Listing 3.17 it was reordered by the plugin to come after the unsafe assumption.

```
| (E_bind (e, e''), s) ->
  (match xJO_red12 e s with
   | e' ->
     (match xis_value_of_expr e with
      | false -> E_bind (e', e'')
      | _ -> assert false (*  *))
   | _ -> assert false (*  *))
```

Listing 3.17: OCaml swapped assumptions

The simple solution to this is to swap the assumptions, however in some cases that leads to to the issues mentioned above.

3.3.3 Syntactic sugar

Due to the multiple layers of automatic naming developing in this framework directly could be cumbersome. I have defined a few OCaml functions to serve as syntactic sugar. The two broad categories of sugar are syntax to build expressions and schedulers.

- Boxing expressions and computations: let boxe e = E_live_expr (LM_expr e), let boxc f = E_live_expr (LM_comp f)
- An infix bind operator: let (>>=) a b = E_bind (a, b)
- Application: let app a b = E_apply (a, b)
- Fork: let fork a b = app (app (E_constant CONST_fork) a) b
- Round-robin and random schedulers in Listing 3.18 and Listing 3.19 respectively.

```
let rec makerr1 () = Seq(S_First, makerr2)
and makerr2 () = Seq(S_Second, makerr1)

let rec evalrr1 e n = (match n with
          | 0 -> e
          | m -> evalrr2 (xJO_red12 e (makerr1 ())) (m-1))
and evalrr2 e n = (match n with
          | 0 -> e
          | m -> evalrr1 (xJO_red12 e (makerr2 ())) (m-1))
```

Listing 3.18: OCaml round-robin scheduler

```
let rec makerand () = if Random.bool() then Seq(S_First, makerand)
  else Seq(S_Second, makerand)

let rec evalrand e n = (match n with
                      | 0 -> e
                      | m -> evalrand (xJO_red12 e (makerand ())) (
 m-1))
```

Listing 3.19: OCaml random scheduler

Chapter 4

Evaluation

4.1 Theoretical evaluation

There are a number of properties that are expected from this system: monadic laws, process calculus axioms and potentially properties of the fixpoint operator, predicates for pairs, type preservation and progress. These requirements are all phrased as equivalences as mentioned in Section 2.3.

First the semantics of the equivalence $\equiv$ had to be examined. If two expressions are equivalent they should behave the same: for an outside observer there has to be no difference which exact expression is evaluated in a black box computation. One way to capture this is by keeping track of potential effects and side-effects. Effects are the return values of an expression, side-effects are the computation placeholders invoked by the system. This is the reason why I phrased the operational semantics with labelled transitions: the labels are descriptions of the side-effects.

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

$$\mathbf{ret}\ e \gg\!= e' \quad \simeq \quad e'\, e$$

$$\mathit{Live\ expr\ v} \gg\!= \mathbf{ret} \quad \simeq \quad \mathit{Live\ expr\ v}$$

$$(\mathit{Live\ expr\ v} \gg\!= f) \gg\!= g \quad \simeq \quad \mathit{Live\ expr\ v} \gg\!= (\lambda x.(f\ v) \gg\!= g)$$

Fork commutativity

Outline of fork commutativity

$$a \,|\, b \quad \simeq \quad b \,|\, a$$

Fork associativity

Fork in this implementation is not associative:

Deadlock properties

$$\delta \,|\, x \quad \simeq \quad x$$
$$\delta \gg\!= e \quad \simeq \quad \delta$$

Remarks on congruence

Type preservation?

Progress?

4.2 Performance

To evaluate the performance of the framework I followed Deleuze[11]. He provides a library for evaluating lightweight and heavyweight threading implementations in OCaml. This evaluation library contains seven implementations of concurrency primitives.

1. **sys**: A heavyweight implementation with system threads.
2. **vm**: A lightweight concurrency implementation based on the `thread` library of OCaml. This implementation is only available in bytecode.
3. **cont**: A continuation monad based lightweight implementation[11, p. 12-13], without verification. Note, this is a much lighter implementation compared to the one in this project.

4. **promise**: A promise monad based implementation of lightweight concurrency[11, p. 13-15].

5. **tramp**: A lightweight implementation based on the trampolined style[11, p. 11-12].

6. **lwt**: An LWT[2] based implementation of lightweight concurrency.

7. **equeue**: An event-based programming style lightweight concurrency implementation[11, p. 15-18] based on the **Equeue** library of OCamlNet. I did not use this implementation in the evaluation as I could not find the right libraries to compile this.

4.2.1 Method

Deleuze provides three examples to evaluate, based on the examples used in Kahn's process language paper[15]. I have implemented the examples in my system. I measured the runtime and memory use of both my implementation and the library implementations for various input sizes. The runtime was measured with *Unix.time* and the memory use was measured with the quick stat function of the `Gc` module of OCaml which gives an interface to the garbage collector. Both bytecode and native code versions were measured. All programs were measured as both OCaml byetcode and native code. All examples were run on an Intel i7-2720QM CPU that is a 4 core hyperthreaded system. This computer had 8 GBs of memory, OCaml version 3.12.1, a linux kernel version 3.11.0-15-generic, caml-shift version of August 2013.

My conjecture was that my system will perform with similar characteristics as lightweight systems, however with a serious overhead due to the complex pattern matching and constructor based system.

4.2.2 Examples

The three examples provided by Deleuze are all process networks from Kahn[15]. A process network is a set of independent processes which communicate through message variables only.

A message variable (mvar) is a shared reference cell that blocks in two cases: if a thread tries to read an empty cell or if a thread tries to put something in a filled cell. A read from an mvar consumes the contents of the variable. The scope of this project did not include formalisation of communication procedures like message variables, however it is simple to implement within the framework. Access to a

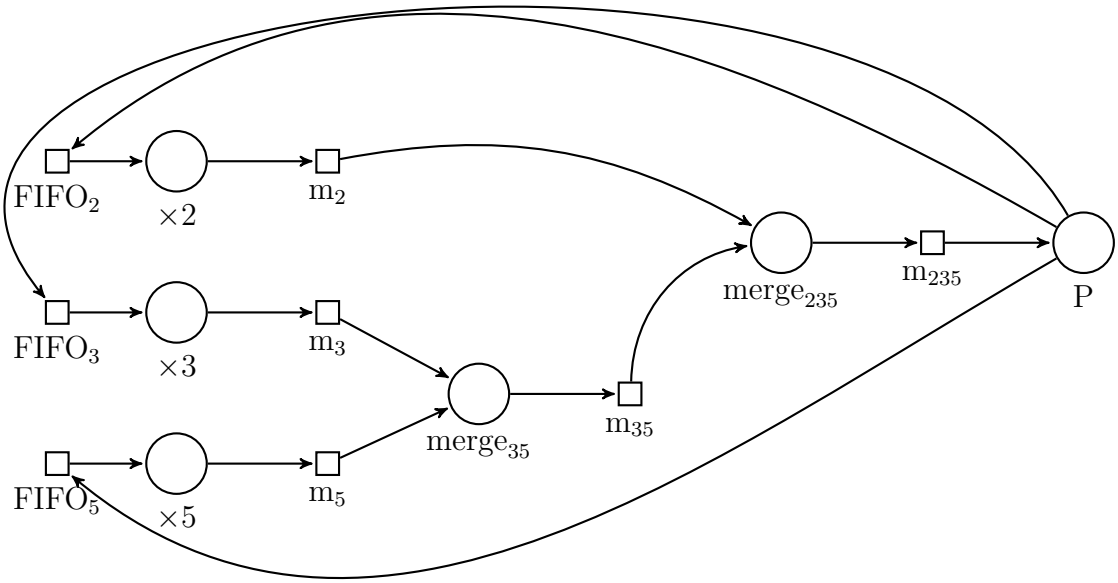


Figure 4.1: `kpn` process outline

message variable is atomic as no two threads run in parallel, even though they are all concurrent. Furthermore, Deleuze used message First-in-First-out queues where putting information to the message FIFO never blocks.

In Figures 4.1, 4.3 and 4.6 processes are denoted with a circle and message variables and FIFOs are denoted by squares. Solid lines denote data flow and dashed lines mean process creation.

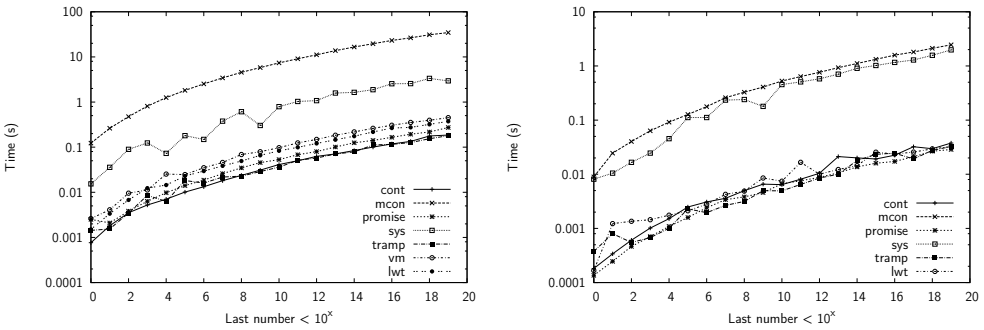
1. `kpn`: A process network calculating all positive integers of the form $2^a 3^b 5^c$ for all a, b, c non-negative integers

2. `sieve`: The well known sieve of Eratosthenes.

3. `sorter`: Concurrent sort of a list of integers.

Kahn process network

`Kpn` is a process network of 6 static threads shown in Figure 4.1:



(a) Bytecode (b) Native code

Figure 4.2: `kpn` execution time

- $\times 2$, $\times 3$, $\times 5$ multiply their input (taken from the corresponding FIFO) by 2, 3 and 5 respectively.
- merge_{35} merges the output of $\times 3$, $\times 5$ using the corresponding message variables: it outputs the lower and fetches a new element that edge.
- Similarly, merge_{235} merges the output of merge_{35} and $\times 2$
- P prints the numbers coming in and copies them to the three FIFOs at the beginning.

This network exemplifies a common case in concurrency: a low number of static threads with simple behaviour. In line with the expectation Figure 4.2 shows that my implementation exhibits exactly the same behaviour as other lightweight implementation, but with a serious overhead: up to 200 times slower than the fastest implementation, **tramp**.

The memory requirement of this system is uninteresting in all implementations: as the process structure is static, so is the memory use.

Sieve of Eratosthenes

The sieve of Eratosthenes is one of the simplest algorithms to find primes:

1. Start with number 2 and have all numbers greater or equal to 2 not crossed out.
2. Take the lowest not crossed out number: it is a prime.
3. Cross out all numbers divisible by the prime just found.

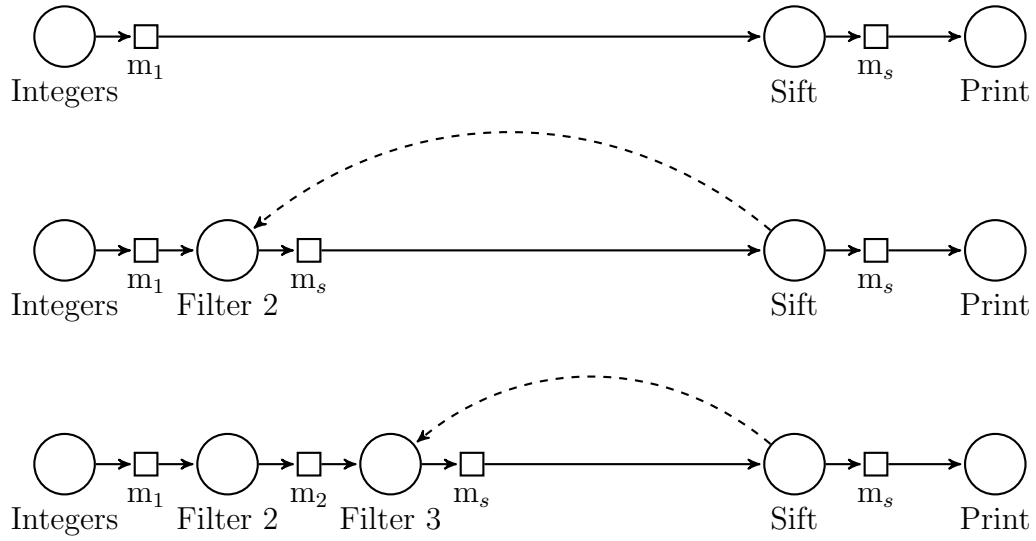


Figure 4.3: `sieve` process outline

4. Repeat from step 2.

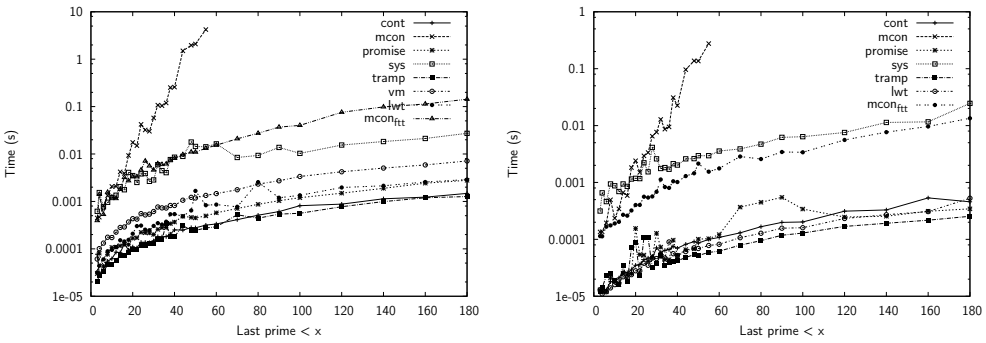
This algorithm is implemented using a dynamic process network. Initially the network is made up of four types of processes:

1. Integers: generates the infinite sequence of integers starting from 2
2. Sift: If a number p is read Sift puts p forward to the printer process and it creates a new process, Filter p and inserts it between the input of Sift and Sift itself in the flow of numbers as shown in Figure 4.3.
3. Filter p discards all numbers divisible by p and passes everything else on.
4. Print just prints all numbers consumed.

This problem is a good example of dynamic thread creation that most general purpose concurrency frameworks should support.

Unlike the expectation, the concurrency framework with random reductions (mcon) in Figure 4.4 exhibits an exponential behaviour. The reason for this is in the way random selection parameters relate to the **fork** tree formed by the expression.

Uniform random reductions assigns equal probabilities to each edge of a fork as it can be seen in Figure 4.5a. In general the probability that the filter process of the kth prime fires is $\frac{1}{2^{k+1}}$. This uneven probability distribution means that for the expected number of reduction steps taken by the system before finding



(a) Bytecode (b) Native code

Figure 4.4: `sieve` execution time

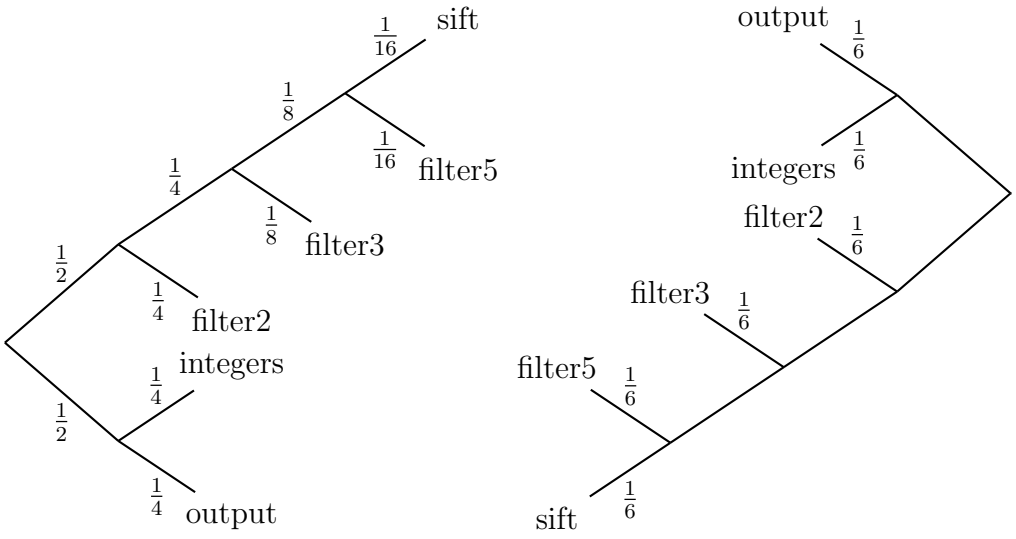
the kth prime is bigger than $2^{k+1} \in O(2^k)$ as it has to make at least one step within each filter.

However, the concurrency framework does not prescribe any evaluation strategy. It is possible to keep track of the fork tree at runtime and equalize the scheduling probabilities of all processes with every computation placeholder. This results in the scheduling probabilities shown in Figure 4.5b. These probabilities reduce the expected number of steps for the kth prime to $k(k+1)$. The speedup is easily seen on Figure 4.4 with the series mcon_{ftt}.

Concurrent sort

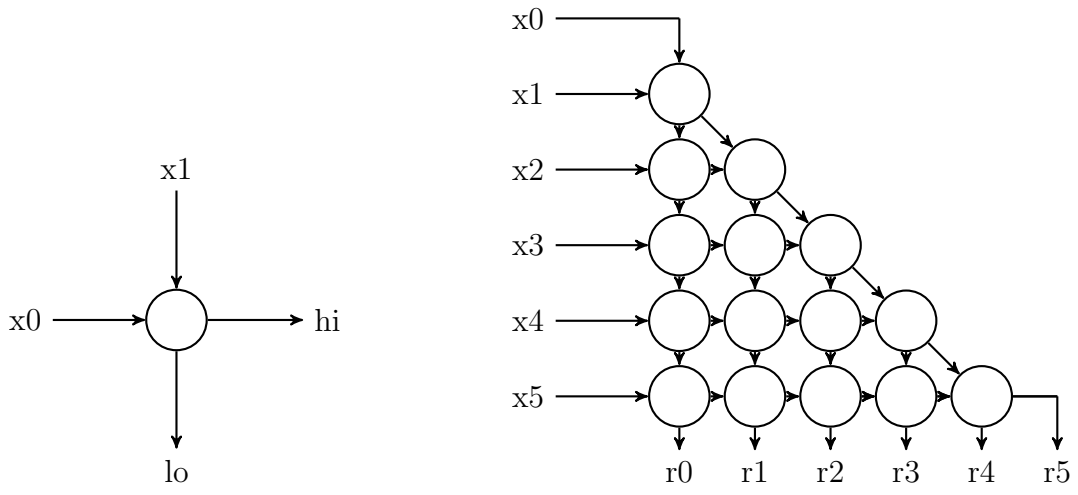
The last example is concurrent sort. Sorting of a list of integers can be done by single comparator units as shown in Figure 4.6a which takes in two values and outputs the lower on one edge and the higher on the other. To sort an entire list we can arrange these elements in a network shown in Figure 4.6b. This network takes in the n element list $x_0, x_1, \ldots, x_n$ and outputs the result $r_0, r_1, \ldots, r_n$. For an n element list the network has $\frac{n(n-1)}{2}$ nodes. Notice that bubble sort and insertion sort are two scheduling of this network. For simplicity I did not show the message variables in Figure 4.6 but they are used on each arrow. This problem is a good example of a task best suited for lightweight concurrency: a very large number of simple threads statically allocated. For a list size of 500, there are 124750 threads.

As expected, my system performed with the same characteristics as other lightweight concurrency solutions, but with a high overhead. Interestingly, the **vm** lightweight implementation shows very bad behaviour, however the heavyweight **sys** implementation performs better with lists longer than 250 elements.



(a) Random without fork tree tracking (b) Fork tree tracking

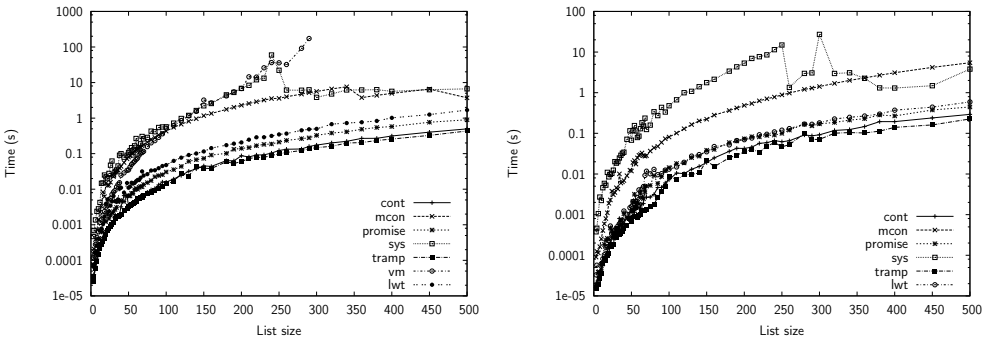
Figure 4.5: Sieve fork tree



(a) Single comparator (b) Full sorting network

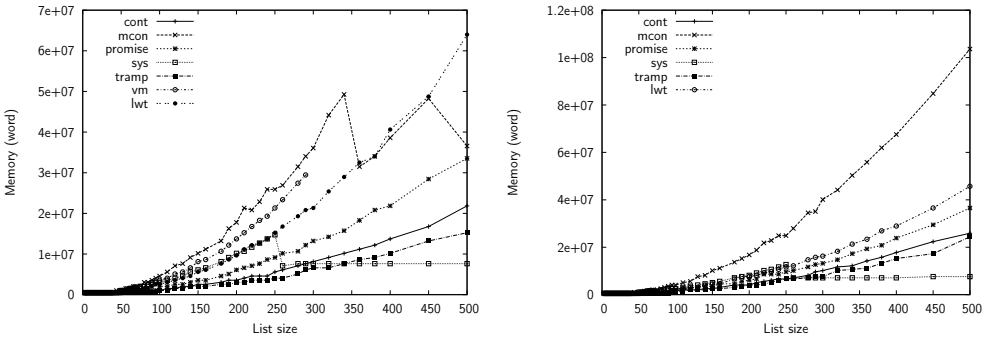
Figure 4.6: `sorter` process outline

The memory use of my system is much higher than the other implementations as shown in Figure 4.8.



(a) Bytecode (b) Native code

Figure 4.7: `sorter` execution time



(a) Bytecode (b) Native code

Figure 4.8: `sorter` memory use

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://www.cl.cam.ac.uk/~so294/ocaml/`, 2008.

[5] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The coq proof assistant. `http://coq.inria.fr/`.

[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] Gilles Kahn, David MacQueen, et al. Coroutines and networks of parallel processes. 1976.

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

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

[18] Oleg Kiselyov. Delimited control in ocaml, abstractly and concretely. *Theoretical Computer Science*, 435:56–76, 2012.

[19] Xavier Leroy. Ocaml-callcc: call/cc for ocaml (2005). `http://pauillac.inria.fr/~xleroy/software.html#callcc`.

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

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

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

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

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

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

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

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

[28] 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.

[29] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott, a tool for writing definitions of programming languages and calculi. `http://www.cl.cam.ac.uk/~pes20/ott/`.

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

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

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

[33] 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.

[34] 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

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

	$expr\ expr'$	
	$expr >>= expr'$	
	$\mathbf{function}\ value\_name : typexpr \rightarrow expr$	bind $value\_name$ in $expr$
	$\mathbf{fix}\ e$	
	$Live\ lm$	
	$\{e, e'\}$	
	$(expr)$	S
	$\mathbf{left}\ e$	
	$\mathbf{right}\ e$	
	$Case\ e_1\ of\ \mathbf{left}\ x_1 \Rightarrow e_2 \mid \mathbf{right}\ x_2 \Rightarrow e_3$	
	$\{v/x\}e$	M
	$\{(\mathbf{fix}\ (functionx : t \rightarrow e))/x'\}e$	M

$value,\ v$	::=	
	$constant$	
	$\mathbf{function}\ value\_name : typexpr \rightarrow expr$	
	$Live\ lm$	
	$\mathbf{left}\ v$	
	$\mathbf{right}\ v$	
	$\{v, v'\}$	
	(v)	S

$select,\ s$	::=	
	1	
	2	

Γ	::=	
	$\mathbf{empty}$	
	$\Gamma, value\_name : typscheme$	

$formula$	::=	
	$judgement$	
	$\mathbf{not}\ (formula)$	
	$typscheme > t$	
	$typscheme = typscheme'$	
	$value\_name = value\_name'$	

$terminals$	::=	
	$\rightarrow$	
	$\mathbf{function}$	
	$\vdash$	
	$\longrightarrow$	

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

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

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

judgement ::=
| | *Jtype*
| | *Jop*

user_syntax ::=
| | *value_name*
| | *label*
| | *ident*
| | *index*
| | *typvar*
| | *typconst*
| | *typexpr*
| | *typscheme*
| | *constant*

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

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

$$\frac{}{\mathit{value\_name} : \mathit{typscheme}\ \mathbf{in}\ \Gamma, \mathit{value\_name} : \mathit{typscheme}}\ \textsc{VTSin\_vn1}$$

$$\frac{\begin{array}{c}\mathit{value\_name} : \mathit{typscheme}\ \mathbf{in}\ \Gamma \\ \mathbf{not}\ (\mathit{value\_name} = \mathit{value\_name}')\end{array}}{\mathit{value\_name} : \mathit{typscheme}\ \mathbf{in}\ \Gamma, \mathit{value\_name}' : \mathit{typscheme}'}\ \textsc{VTSin\_vn2}$$

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

$$\frac{}{\Gamma \vdash \mathbf{ret} : t \to \mathbf{con}\ t}\ \textsc{constant\_ret}$$

$$\frac{}{\Gamma \vdash \mathbf{fork} : (\mathbf{con}\ t_1) \to ((\mathbf{con}\ t_2) \to (\mathbf{con}\ ((t_1 * (\mathbf{con}\ t_2)) + ((\mathbf{con}\ t_1) * t_2))))}\ \textsc{constant\_fork}$$

$$\frac{}{\Gamma \vdash \mathbf{unit} : \mathbf{tunit}}\ \textsc{constant\_unit}$$

$$\frac{}{\Gamma \vdash \mathbf{stop} : t}\ \textsc{constant\_stop}$$

$$\frac{}{\Gamma \vdash \mathbf{pair} : t_1 \to (t_2 \to (t_1 * t_2))}\ \textsc{constant\_pair}$$

$$\frac{}{\Gamma \vdash \mathit{proj}1 : (t_1 * t_2) \to t_1}\ \textsc{constant\_proj1}$$

$$\frac{}{\Gamma \vdash \mathit{proj}2 : (t_1 * t_2) \to t_2}\ \textsc{constant\_proj2}$$

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

$$\frac{\begin{array}{c}x : \mathit{typscheme}\ \mathbf{in}\ \Gamma \\ \mathit{typscheme} > t\end{array}}{\Gamma \vdash x : t}\ \textsc{Get\_value\_name}$$

$$\frac{\Gamma \vdash \mathit{constant} : t}{\Gamma \vdash \mathit{constant} : t}\ \textsc{Get\_constant}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : t_1 \to t_2 \\ \Gamma \vdash e' : t_1\end{array}}{\Gamma \vdash e\ e' : t_2}\ \textsc{Get\_apply}$$

$$\frac{\Gamma, x_1 : (\,)t_1 \vdash e : t}{\Gamma \vdash \mathbf{function}\ x_1 : t_1 \to e : t_1 \to t}\ \textsc{Get\_lambda}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathit{Live}\ \mathit{exp}\ e : \mathbf{con}\ t}\ \textsc{Get\_live\_exp}$$

$$\frac{}{\Gamma \vdash \mathit{Live}\ (\mathit{comp}\ \mathit{lab}) : \mathbf{con}\ \mathbf{tunit}}\ \textsc{Get\_live\_comp}$$

$$\frac{\Gamma \vdash e : t \to t}{\Gamma \vdash \mathbf{fix}\, e : t} \quad \textsc{Get\_fix}$$

$$\frac{\Gamma \vdash e : \mathbf{con}\, t \quad \Gamma \vdash e' : t \to \mathbf{con}\, t'}{\Gamma \vdash e >>= e' : \mathbf{con}\, t'} \quad \textsc{Get\_bind}$$

$$\frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash e' : t_2}{\Gamma \vdash \{e, e'\} : (t_1 * t_2)} \quad \textsc{Get\_pair}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathbf{left}\, e : t + t'} \quad \textsc{Get\_TInl}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathbf{right}\, e : t' + t} \quad \textsc{Get\_TInr}$$

$$\frac{\Gamma \vdash e : t + t' \quad \Gamma, x : ()t \vdash e' : t'' \quad \Gamma, x' : ()t' \vdash e'' : t''}{\Gamma \vdash Case\, e\, of\, \mathbf{left}\, x \Rightarrow e' | \mathbf{right}\, x' \Rightarrow e'' : t''} \quad \textsc{Get\_TCase}$$

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

$$\frac{}{(\mathbf{function}\, x : t \to e)\, v \xrightarrow[s]{\tau} \{v/x\}e} \quad \textsc{JO\_red\_app}$$

$$\frac{e \xrightarrow[s]{rl} e''}{(\mathbf{fork}\, (Live\, exp\, e))\, (Live\, lm) \xrightarrow[1]{rl} (\mathbf{fork}\, (Live\, exp\, e''))\, (Live\, lm)} \quad \textsc{JO\_red\_forkmove1}$$

$$\frac{e' \xrightarrow[s]{rl} e''}{(\mathbf{fork}\, (Live\, lm))\, (Live\, exp\, e') \xrightarrow[2]{rl} (\mathbf{fork}\, (Live\, lm))\, (Live\, exp\, e'')} \quad \textsc{JO\_red\_forkmove2}$$

$$\frac{}{(\mathbf{fork}\, (Live\, exp\, v))\, (Live\, lm) \xrightarrow[1]{\tau} Live\, exp\, (\mathbf{left}\, (\{v, (Live\, lm)\}))} \quad \textsc{JO\_red\_forkdeath1}$$

$$\frac{}{(\mathbf{fork}\, (Live\, lm))\, (Live\, exp\, v') \xrightarrow[2]{\tau} Live\, exp\, (\mathbf{right}\, (\{(Live\, lm), v'\}))} \quad \textsc{JO\_red\_forkdeath2}$$

$$\frac{}{(\mathbf{fork}\, (Live\, (comp\, lab)))\, (Live\, lm) \xrightarrow[1]{lab} Live\, exp\, (\mathbf{left}\, (\{\mathbf{unit}, (Live\, lm)\}))} \quad \textsc{JO\_red\_forkdocomp1}$$

$$\frac{}{(\mathbf{fork}\, (Live\, lm))\, (Live\, (comp\, lab)) \xrightarrow[2]{lab} Live\, exp\, (\mathbf{right}\, (\{(Live\, lm), \mathbf{unit}\}))} \quad \textsc{JO\_red\_forkdocomp2}$$

$$\frac{}{\mathbf{ret}\, v \xrightarrow[s]{\tau} (Live\, exp\, v)} \quad \textsc{JO\_red\_ret}$$

$$\frac{e \xrightarrow[s]{rl} e'}{e >>= e'' \xrightarrow[s]{rl} e' >>= e''} \quad \textsc{JO\_red\_evalbind}$$

$$\frac{e \xrightarrow[s]{rl} e'}{(Live\, exp\, e) >>= e'' \xrightarrow[s]{rl} (Live\, exp\, e') >>= e''} \quad \textsc{JO\_red\_movebind}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A.1 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[32] 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[29].

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[34].

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[5] 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 [26, 4], 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.