Tamás Kispéter

Monadic Concurrency in OCaml

Part II in Computer Science

Churchill College

May 14, 2014

Proforma

Name:	**Tamás Kispéter**
College:	**Churchill College**
Project Title:	**Monadic Concurrency in OCaml**
Examination:	**Part II in Computer Science, July 2014**
Word Count:	**12000**[1]
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

The semantics were defined in the file `mconbase.ott`. These semantics were translated to a representation in Coq in `mconbase.v`. The representation was modified for extraction in the file `mconextract.v`. Supporting proofs are in files `progress.v`, `weakBisimulations.v`, `forkProofs.v`, `redTotalDetProp.v`, `forkDeadlock.v`, `mconbaseMonProofs.v`. The semantics were hand extracted to the file `combinedPieces.ml`, made runnable in `mconbase.ml`. The OCaml code was evaluated using the files

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

`sugarcube.ml`, `mvars.ml`, `kpn.ml`, `sieve.ml`, `sorter.ml`. All other files are libraries used in the project.

Special Difficulties

There were no special difficulties in the completion of this project.

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 14, 2014

Contents

List of Figures

Listings

List of Theorems

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 are discovered in libraries considered safe. Some of these bugs even extensive unit testing could not find. As Dijkstra said,

> "Testing shows the presence, not the absence of bugs."

On the other hand, verification can show the absence of bugs, at least with respect to the formal model of the system.

Motivation of the project is to implement a verified concurrency framework in OCaml. Verified concurrent systems have been researched for languages including C[39], C++ and Java[31], 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[43]. They both provide 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[45] 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[32] implements join calculus over OCaml, Functory[23] focuses on distributed computation, OCamlNet exploits multiple cores and OCamlMPI[27] 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[17] are full[18, 26] and delimited[24] continuations[41], trampolined style[19], continuation monads[15], promise monads[30] and event based programming (as used, for example in the OCamlNet[42] project). This work focuses on the continuation monad style. Monads will be detailed in Section 2.3.

All concurrency styles have at least one base concurrent primitive that achieves concurrent behaviour. This primitives 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, 12, 11, 28]. Coq[5], HOL and Isabelle are widely used proof assistants. Tools like Ott[40] 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 relations. This representation is more amenable to proofs than to actual execution. Letouzey[29] provides methods to extract a large subset of the logical representation from Coq to executable OCaml or Haskell code. Berghofer[7] offers similar extraction methods for Isabelle and HOL, but only to ML.

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. This algebra typically comes with a number of operations and axioms. For example, Milner[33] describes the concurrency primitives:

- $P \mid Q$ 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
- $P + Q$ for summation, or choice between two processes, P and Q

Axioms include:

- $P \mid Q \equiv Q \mid P$: parallel composition is commutative
- $P + P \equiv P$: choice is idempotent

This project aimed to have simple but powerful operational semantics. Simplicity is required in both the design and the interface. There is a short timespan for implementation, but 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[17] 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[26] explores call/cc style semantics 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[24, 25] developed an OCaml library for delimited continuations and advocates it over call/cc style.

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

Trampolined style, as Ganz[19] 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[15], while promise monads[30] 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[20] 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. Monads are a popular approach to organising code that may have side-effects. 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 $\mathbf{con}\,T$ where T is the type parameter. Note **con** is an arbitrary name, a 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 $\textbf{ret} : \forall\alpha.\alpha \rightarrow \textbf{con}\,\alpha$.

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

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

1. ret is a left neutral element of bind:

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

2. ret is a right neutral element of bind:

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

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.

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.

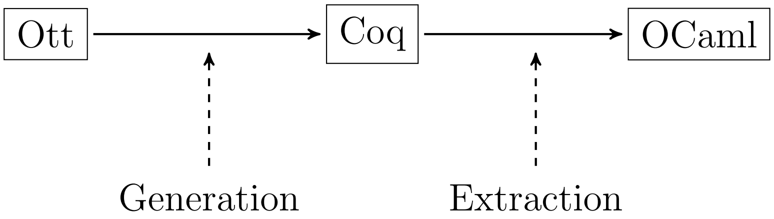


Figure 2.1: High level view of the tool chain

The high level view of the relationship between these tools is shown in Figure 2.1. This view is somewhat simplified from the actual chain that I return to in Figure 3.1 in Chapter 3.

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

2.4.1 Ott

To avoid duplication of the semantics in several formats I have to chosen to use a supporting tool called Ott[40]. 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.

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 2.1: Ott grammar example

In Listing 2.1 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 string "t_" in translation to have non-ambiguous names. The right hand side of each line in Listing 2.1 is a so called homomorphism. A homomorphism describes a manual translation to a target language. It starts with a destination like ocaml for OCaml,tex or com for LaTeX and ich for Isabelle, Coq and HOL. An expression in double square brackets [[t]] stands for the translation of the expression within the brackets.

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

The variable x in Listing 2.1 is a metavariable. 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 2.2: Ott metavariable definition

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

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

subrules
  v <:: t
```

Listing 2.3: Ott value subgrammar example

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

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

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

Listing 2.4: Ott substitution example

The statement single t x :: tsubsts in Listing 2.4 defines a single substitution function called tsubsts_t over terms defined by the grammar for t and for variables represented by the metavariable x. This is the same tsubst_t mentioned Listing 2.1.

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 in Listing 2.1. 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.

```
∀ 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.

```
∀ 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(Listing 2.10) and co-inductively(Listing 2.11). Co-inductive data structures represent potentially infinite data. They are supported by a number of programming languages, since they are useful for defining data structures such as lazy lists, trees and streams.

```
Inductive seq : nat → Set :=
  | niln  : seq 0
  | consn : ∀ 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 co-inductive 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 : ∀ (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 applications 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.

As Letouzy[29] explains, Coq provides 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. These are most often general relations where the "truth" of the proposition means membership in the relation. The addition relation in Listing 2.16 of three elements is one such logical inductive construct.

```
Inductive add : nat → nat → nat → Prop :=
| addO : ∀ n , add n O n
| addS : ∀ 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, meaning there may not be any subset of arguments of the relation that a function can map to the rest of the arguments. Furthermore, 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 [16, 44] by marking the input/output modalities of the inductively defined proposition I can extract certified programs. 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 supports a wide range of functional 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 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 2.19: 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 2.20: 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 2.21: 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. 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 so that I can evaluate the correctness of the system. For performance evaluation I used an evaluation framework developed by Deleuze for his paper comparing lightweight concurrency implementations in OCaml[17].

Chapter 3

Implementation

For simplicity, I call the concurrency framework MOCaml. MOCaml was implemented as a small language of expressions and a reduction relation. These two were then translated to OCaml as a set of type definitions and an evaluation function respectively.

MOCaml supports a number of features: simple sequencing, concurrent execution, recursion, casing and user supplied scheduling. Furthermore, with simple syntactic sugar it is possible to extend the framework by safe shared communication channels and complex scheduling decisions.

Listing 3.1 is an example of the framework in use: the user has defined two threads, increment and print, that run concurrently. The thread increment adds one to a shared reference cell while print prints it. These two threads are evaluated with a random scheduling algorithm.

The general development workflow is as follows: a MOCaml user would build an expression within the language with the provided syntactic sugar by for example boxing up a user computation with boxc on Line 2 or forming the parallel composition of two threads with fork on Line 4. The built expression is then passed to a scheduler as on Line 23. The user can choose to use a provided scheduler like the infinite random scheduler defined on Lines 12 to 15 or build his own using the single step evaluation function xJO_red12.

```
(* Syntactic sugar *)
let boxc f = E_live_expr (E_comp f)
let ( >>= ) a b = E_bind (a, b)
let fork a b = E_fork (E_live_expr a, E_live_expr b)
let cunit = E_unit
let func v t e = E_function (v, t, e)
let forever_compute e v1 v2 =
    E_fix (func v1 TE_unit ((boxc e) >>=
       (func v2 TE_unit (E_ident (v1)))))

(* Scheduler *)
let rec makerand () = if Random.bool()
                       then Seq(D_Left, makerand)
                       else Seq(D_Right, makerand)
let rec runForever e = (runForever (xJO_red12 e (makerand ())))

(* User code *)
let increment n =  forever_compute (fun _ -> n:= !n +1; cunit ) 1 2
let print n =  forever_compute
      (fun _ ->  print_int !n; print_string"\n"; cunit) 1 2

let run () = let n = ref 1 in
  runForever (fork (increment n)  (print n))
```

Listing 3.1: A simple example in MOCaml

The implementation of MOCaml was done in 5 stages as shown in Figure 3.1.

Stage 1. I defined the overall semantics in Ott.

Stage 2. Ott generated a Coq representation of the semantics.

Stage 3. I manually modified the Coq representation to be suitable for the extraction plugin detailed in Section 2.4.2.

Stage 4. The extraction plugin and the built-in extraction facilities in Coq extracted OCaml code faithful to the Coq representation.

Stage 5. I modified the OCaml code to be runnable and provided some syntactic sugar.

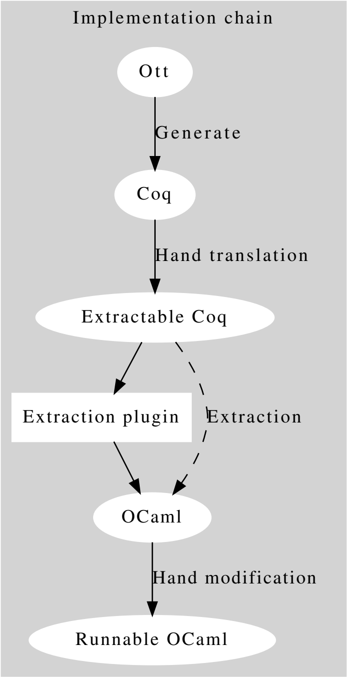


Figure 3.1: Detailed outline of the implementation

3.1 The semantics

In this section I describe the implemented semantics. These semantics were written in Ott. Much of the inspiration for the semantics of the sequential features comes from Benjamin C. Pierce: *Types and Programming Languages*[35]. The Ott code was based on the simply typed λ-calculus[14]. For a LaTeX version of the full semantics as produced by Ott see appendix A.

The semantics splits into three main parts

1. Grammars:
   - expression grammar, denoted by e
   - value subgrammar, denoted by v
2. Types and type judgements
3. Reduction judgements

I detail the semantics in two groups of features: sequential features that form a standard extension to the simply typed λ-calculus and the concurrency features. The basic features are arrow types (functions), sum types (tagged unions), product types (pairs) and the fixpoint combinator. The monadic concurrency features are the monadic primitives and the fork operator. Each feature group is presented by first examining the syntax and typing rules of the features followed by their operational semantics. In the syntax parts I introduce every new syntactic form used in the evaluation and typing rules, but I do not repeat previously mentioned syntax. The presentation of both the syntax and the evaluation and typing rules were slightly simplified for better readability.

3.1.1 Sequential features: arrow, sum and product types, fixpoint

The type judgement relation

Syntax

e ::=		*expressions:*
	x	*variable*
	$()$	*unit*
v ::=		*values:*
	$()$	*unit*
T ::=		*types:*
	unit	*unit*
Γ ::=		*contexts:*
	$\emptyset$	*empty context*
	$\Gamma, x : T$	*term variable binding*

Typing $\boxed{\Gamma \vdash e : T}$

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

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

Figure 3.2: Syntax of the type judgement with the example of **unit**

As MOCaml is a simply typed language: that is to say that each well formed expression has one or more type. The typing judgement is a relation relating expressions to types, built inductively by typing rules with no quantification.

MOCaml, like many other languages, supports variables. The type of a variable is decided by a further piece of information, the typing environment Γ. The typing environment is a variable-to-type function, often written as a list of variable-type pairs. The type judgement relation is therefore a triple (Γ, e, T)

where Γ is the typing environment, e is an expression and T is its type. This triple is often written as $\Gamma \vdash e : T$.

Two simple examples of typing judgements are the variable typing rule (T-Var) and the **unit** typing rule (T-Unit) in Figure 3.2. Variables get their types from the typing environment Γ and the expression () has type **unit** in all environments.

Arrow types

Syntax

$$\begin{array}{lll} e ::= & \ldots & \textit{expressions:} \\ & \lambda x : T.e & \textit{abstraction} \\ & e\,e & \textit{application} \\ & & \\ T ::= & \ldots & \textit{types:} \\ & T \to T & \to \textit{type} \end{array}$$

Typing $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash \lambda x : T.e : T \to T'} \quad \text{(T-Abs)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T \to T' \\ \Gamma \vdash e' : T'\end{array}}{\Gamma \vdash e\,e' : T'} \quad \text{(T-App)}$$

Figure 3.3: Syntax and typing of arrow types

Arrow types or functions are ubiquitous in functional programming languages. In Figure 3.3 I detail the basic syntax and typing of functions in a simply typed setting, while Figure 3.8 shows the operational semantics. The style and details are based on Pierce[35, 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. If expression e has type T' given that variable x has type T, then the λ-abstraction $\lambda x : T.e$ has type $T \to T'$ (T-App). If I pass an expression e' with type T to an expression e that has the arrow type $T \to T'$ then the application itself will have type T' (T-Abs) as e will return a result of type T'.

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.

Figure 3.4 shows the basic structure of sum types that is based on Pierce[35, p. 132]. 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. As shown in (T-Left) and

Syntax

$e ::= \dots$	*terms:*
$\textbf{left}\, e$	*left*
$\textbf{right}\, e$	*right*
$\textbf{case}\ e\ \textbf{of}$ $\quad \textbf{left}\ x_1 \Rightarrow e_1$ $\quad \mid \textbf{right}\ x_2 \Rightarrow e_2$	*case*
$T\ ::= \dots$	*types:*
$T + T'$	*sum*

Typing $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash\ e : T}{\Gamma \vdash \textbf{left}\, e\ :\ T + T'} \quad \text{(T-Left)}$$

$$\frac{\Gamma \vdash\ e : T'}{\Gamma \vdash \textbf{right}\, e\ :\ T + T'} \quad \text{(T-Right)}$$

$$\frac{\begin{array}{c}\Gamma \vdash\ e : T + T' \\ \Gamma, x_1 : T \vdash\ e_1\ :\ T'' \\ \Gamma, x_2 : T' \vdash\ e_2\ :\ T''\end{array}}{\Gamma \vdash (\textbf{case}\ e\ \textbf{of}\ \textbf{left}\ x_1 \Rightarrow e_1 \mid \textbf{right}\ x_2 \Rightarrow e_2\ )\ :\ T''} \quad \text{(T-Case)}$$

Figure 3.4: Syntax and typing of sum types

(T-Right) these variants will have type $T + T'$ for some type T' and T respectively. A labelled expression can then be destructed by the expression "**case** e **of left** $x_1 \Rightarrow e_1 \mid$ **right** $x_2 \Rightarrow e_2$ ". The typing rule (T-Case) gives the **case** expression the common type, T'', of e_1 and e_2 when assuming $x_1\ :\ T$ and $x_2\ :\ T'$ respectively. Pierce gives a slightly more complex version of sum types that ensures that every expression has a unique type. For this project I used the version given here for simplicity, but further work should include a change to the uniquely typable variant.

In this project I use sum types in the signature of **fork** that will be detailed in Section 3.1.2.

Product types

A second 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 e and e' have types T and T' respectively the type of the pair $\{e,\ e'\}$ is written $T \star T'$ as described in rule (T-Pair). I introduced two primitive functions to deal with pairs: $\textbf{proj}_1$ takes the first element of a pair and $\textbf{proj}_2$ takes the second. If a pair has type $T \star T'$ the projections will have types T and T' by the rules (T-Proj1) and (T-Proj2) respectively.

Syntax

e ::=	$\ldots$	*terms:*
	$\{e, e'\}$	*pair*
	$\mathbf{proj}_1\, e$	*first projection*
	$\mathbf{proj}_2\, e$	*second projection*
T ::=	$\ldots$	*types:*
	$T \star T'$	*product type*

Typing $\boxed{\Gamma \vdash e : T}$

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

$$\frac{\Gamma \vdash e : T \star T'}{\Gamma \vdash \mathbf{proj}_1\, e : T} \quad \text{(T-Proj1)}$$

$$\frac{\Gamma \vdash e : T \star T'}{\Gamma \vdash \mathbf{proj}_2\, e : T'} \quad \text{(T-Proj2)}$$

Figure 3.5: Syntax and typing of product types

Figure 3.5 shows the syntax and typing rules of product types defined in this language. The style and details are based on Pierce[35, p. 126].

Fixpoint combinator

Syntax

e ::=	$\ldots$	*terms:*
	$\mathbf{fix}\, e$	*fixpoint*

Typing $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash e : T \to T}{\Gamma \vdash \mathbf{fix}\, e : T} \quad \text{(T-Fix)}$$

Figure 3.6: Syntax and typing of the fixpoint operator

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 fixpoints with the syntax and typing properties shown in Figure 3.6. The **fix** primitive is based on the idea, that if a function f is supplied that takes “the rest of the computation” and it can prefix a step, then by assuming that the **fix** of f is “the rest of the computation” it is simple to show that **fix** must have type $(T \to T) \to T$ as shown in the rule (T-Fix). It is important to note that because of (T-Fix) all types have at least one term in them:

$$\textsc{T-Fix}\ \dfrac{\textsc{T-Abs}\ \dfrac{\textsc{T-Var}\ \dfrac{x : T \in \Gamma,\ x : T}{\Gamma,\ x : T \vdash x : T}}{\Gamma \vdash \lambda x : T.x : T \to T}}{\Gamma \vdash \mathbf{fix}\,(\lambda x : T.\, x) : T} \tag{T-All}$$

The reduction relation of MOCaml

Syntax		
rl ::=		*labels:*
	τ	*silent*
	e	*action*
d ::=		*decision:*
	L	*left*
	R	*right*
s ::=	$d\ s$	*select*

Evaluation

$$e \xrightarrow[s]{rl} e' \tag{Red}$$

Figure 3.7: Syntax and semantics of the reduction relations

The reduction semantics of MOCaml is given as small step labelled transitions. In small step semantics the reduction relation between two expressions $e \to e'$ means that e can directly and atomically transition to e'.

Labelled transitions further extend the idea of reduction relations by attaching a label to each reduction. In this project I used a 4-tuple (e, s, rl, e') of the starting expression, a selection operator that will be supplied at runtime to pick between potential reductions, a reduction label that describes the observable action and the ending expression of the transition.

As Figure 3.17 in Section 3.1.2 will show, MOCaml has to make a decision at every **fork** it encounters when reducing an expression a step. I wanted the selection(s) to fully describe all decisions(d) that the system has to make. I opted for a co-inductive stream of decisions because there can be an arbitrary number of **fork**s nested in each other with an arbitrary number of decisions necessary to make a single step.

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

Listing 3.2: Coq co-inductive decision sequence

In Listing 3.2 I define `select` with the co-inductive notation introduced in Section 2.4.2, where each element is a pair of a decision and a further `select` representing the rest of the decisions.

The 4-tuple (e, s, rl, e') means that given the selection operator s, a starting expression e can move to expression e' with a side-effect rl. I use the notation $e \xrightarrow[s]{rl} e'$ as in (Red) in Figure 3.7. The rl label may be τ which is a silent action or an atomic action e that can be observed. A silent action τ is considered unobservable from outside MOCaml. The atomic action e is considered an expression within the model. This choice was based on the behaviour of placeholders in Section 3.1.2. The inspiration to use a labelled transition system to model side-effects comes from Milner's "A calculus of communicating systems" book[33].

Operational semantics of the sequential MOCaml features

Values

$$\begin{array}{llr} v ::= & \ldots & \textit{values:} \\ & \lambda x : T.e & \textit{abstraction} \end{array}$$

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

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

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

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

Figure 3.8: Pperational semantics of arrow types

Figure 3.8 shows the transition rules for functions and applications. I used call-by-value semantics for functions and left-to-right reduction. This means that in an application $e\, e'$ first e reduces to a function abstraction by rule (R-App1), then the argument reduces to a value by rule (R-App1). This argument value is then substituted into the function body with the rule (R-Subst). I used two simplifications in logic that might make development slightly harger: arguments of abstractions are explicitly annotated in functions and substitution does not provide facilities for renaming. The lack of renaming is a trade-off between logical simplicity and ease of use and I chose to keep the logical model as simple as possible. It is presumed that the user of the framework takes care of picking fresh variable names, but I provided facilities with the syntactic sugar to pick fresh variables.

Values

$$\begin{array}{lll} v ::= & \ldots & \textit{values:} \\ & \textbf{left}\ v & \textit{left} \\ & \textbf{right}\ v & \textit{right} \end{array}$$

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

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

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

$$\frac{e \xrightarrow[s]{rl} e'}{\begin{array}{c}(\textbf{case}\ e\ \textbf{of left}\ x_1 \Rightarrow e_1 \mid \textbf{right}\ x_2 \Rightarrow e_2\ ) \\ {\scriptstyle s}\big\downarrow{\scriptstyle rl} \\ (\textbf{case}\ e'\ \textbf{of left}\ x_1 \Rightarrow e_1 \mid \textbf{right}\ x_2 \Rightarrow e_2\ )\end{array}} \qquad \text{(R-Case)}$$

$$\frac{}{(\textbf{case left}\ v\ \textbf{of left}\ x_1 \Rightarrow e_1 \mid \textbf{right}\ x_2 \Rightarrow e_2\ ) \xrightarrow[s]{\tau} \{v/x_1\}e_1} \qquad \text{(R-CaseL)}$$

$$\frac{}{(\textbf{case right}\ v\ \textbf{of left}\ x_1 \Rightarrow e_1 \mid \textbf{right}\ x_2 \Rightarrow e_2\ ) \xrightarrow[s]{\tau} \{v/x_2\}e_2} \qquad \text{(R-CaseR)}$$

Figure 3.9: Operational semantics of sum types

In Figure 3.9 I describe the transition rules for constructs related to sum types: the destructor **case** and the constructors **left** and **right**. The tags **left** and **right** reduce their argument until it is a value by rules (R-Left) and (R-Right) respectively. The **case** expression first reduces its argument e to a tagged value by rule (R-Case). Depending on the tag of the value, this is then silently reduced by either (R-CaseL) or (R-CaseR) to e_1 or e_2 respectively with substituting the value within the tag for the corresponding variable. The **case** expression can be used to dynamically exploit some behaviour information and signal within MOCaml without the need to include the complexity of redefining conditional constructs like **if**. Much like in the case of functions, **case** binds the variables x_1 and x_2 within the expressions e_1 and e_2 respectively. As the substitution does not avoid variable capture the user must take care to use fresh variables.

Values

$$\begin{array}{lll} v ::= & \ldots & \textit{values:} \\ & \{v,\ v'\} & \quad \textit{pair} \end{array}$$

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

$$\frac{e \xrightarrow[s]{rl} e''}{\{e, e'\} \xrightarrow[s]{rl} \{e'', e'\}} \ \text{(R-Pair1)} \qquad \frac{e \xrightarrow[s]{rl} e'}{\mathbf{proj}_1\, e \xrightarrow[s]{rl} \mathbf{proj}_1\, e} \ \text{(R-Proj1-Eval)}$$

$$\frac{e' \xrightarrow[s]{rl} e''}{\{v, e'\} \xrightarrow[s]{rl} \{v, e''\}} \ \text{(R-Pair2)} \qquad \frac{e \xrightarrow[s]{rl} e'}{\mathbf{proj}_2\, e \xrightarrow[s]{rl} \mathbf{proj}_2\, e} \ \text{(R-Proj2-Eval)}$$

$$\mathbf{proj}_1\, \{v,\ v'\} \xrightarrow[s]{\tau} v \ \text{(R-Proj1)} \qquad \mathbf{proj}_2\, \{v,\ v'\} \xrightarrow[s]{\tau} v' \ \text{(R-Proj2)}$$

Figure 3.10: Operational semantics of product types

Figure 3.10 details the operational semantics of pairs, the constructor $\{e,\ e'\}$ and destructors $\mathbf{proj}_1$ and $\mathbf{proj}_2$. All three of these constructs first reduce their arguments left-to-right by the rules (R-Pair1) and (R-Pair2) in the case of $\{e,\ e'\}$ and by the rules (R-Proj1-Eval) and (R-Proj2-Eval) for $\mathbf{proj}_1$ and $\mathbf{proj}_2$ respectively. When reduced, $\mathbf{proj}_1$ and $\mathbf{proj}_2$ silently take the left and right values by the rules (R-Proj1) and (R-Proj2) respectively.

These semantics mimic the implementation by Pierce[35, p. 126]. In a previous iteration of these primitives, I provided them as function values. The **pair** primitive was a higher order function, and $\mathbf{proj}_1$ and $\mathbf{proj}_2$ could be passed around as values unapplied. However, Ott did not handle the partial application of the **pair** primitive correctly with respect to the value subgrammar. Therefore, I decided to include them as fully applied primitives. The new semantics actually provide a way to offer the original versions of the primitives by λ-abstraction:

$$\lambda\, x \,:\, T.\lambda\, y \,:\, T'.\{x,\, y\}, \quad \lambda\, x \,:\, T \star T'.\mathbf{proj}_1\, x, \quad \lambda\, x \,:\, T \star T'.\mathbf{proj}_2\, x$$

The above expressions has equivalent behaviour to the original versions of the corresponding primitives.

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

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

$$\frac{e \xrightarrow[s]{rl} e'}{\mathbf{fix}\, e \xrightarrow[s]{rl} \mathbf{fix}\, e'} \qquad \text{(R-Fix-Eval)}$$

Figure 3.11: Operational semantics of the fixpoint operator

In denotational semantics loops and other recursions are treated as the greatest fixpoints of continuous functions. A fixpoint combinator y is defined as a term that given a function f satisfies

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

This axiom poses a requirement on the type of y: it should be $(T \to T) \to T$ which is the type in Figure 3.6. (I return to the exact notion of the equivalence relation $\equiv$ in Section 4.1 for theoretical evaluation.) In untyped λ-calculus there are many simple terms that behave as fixpoints, for example the Y combinator:

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

However in strict, call-by-value languages like MOCaml arguments are always reduced before the function application can begin. In this setting the Y combinator would always diverge. There are few implementation options in call-by-value languages, but I chose to use the one described in Pierce[35, p. 144]. This style fits well with the tool chain I used. Pierce's implementation does not require a partial function application that Ott sometimes handles incorrectly with respect to the value subgrammar. In this implementation **fix** first reduces its argument by the rule (R-Fix-Eval) to a value and if it is a function abstraction **fix** silently substitutes the fixed function into its variable as shown in rule (R-Fix).

3.1.2 Concurrency features: computation place holders, monadic primitives and fork

The standard features described in the previous section form the basis on which I built the primitives for monadic concurrency. In this section I first describe the syntax and types of the primitives and then move on to operational details.

Computation place holders

Syntax

e ::=	...	*terms:*
	comp e	*computation*

Typing $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \textbf{comp}\ e : T} \quad \text{(T-Comp)}$$

Figure 3.12: Syntax and typing of the computation place holder

In previous subsections there were no constructs that accept external computations and MOCaml could not behave as a general purpose concurrency framework without the ability to insert arbitrary computations. In Figure 3.12 I introduce a new expression called a computation place holder. These are the holes that will be filled by actual OCaml **unit** → expr functions to be evaluated.

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. From the monadic axioms, mentioned in Section 2.3, it follows that it has type $T \rightarrow$ **con** T. This type corresponds to the fully applied typing rule (T-Ret) in Figure 3.13.

- $\gg=$, also known as bind, sequences two operations. The second argument is the continuation for the first parameter. More formally it has a type **con** $T \rightarrow (T \rightarrow$ **con** $T') \rightarrow$ **con** T'. Bind takes a boxed up computation and a function that takes the value of the computation and returns a new box. This corresponds to the type shown in rule (T-Bind)

The tag **live** is attached to boxed up expressions, much like **left** and **right** for sum types. A boxed up computation has type **con** T if the expression within has type T, as described in the rule (T-LiveExpr).

Syntax

e ::=	$\ldots$	*terms:*
	live e	*live expression*
	ret e	*return*
	$e \gg= e'$	*bind*
T ::=	$\ldots$	*types:*
	con T	*concurrent*

Typing $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \textbf{live}\ e : \textbf{con}\ T} \quad \text{(T-LiveExpr)}$$

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

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

Figure 3.13: Syntax and typing of the monadic primitives: ret and bind

Fork

Syntax

e ::=	$\ldots$	*terms:*
	fork $e\ e'$	*fork*

Typing $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash e : \textbf{con}\ T \qquad \Gamma \vdash e : T_1 \rightarrow \textbf{con}\ T'}{\Gamma \vdash \textbf{fork}\ e\ e' : \textbf{con}\ ((T \star \textbf{con}\ T') + ((\textbf{con}\ T) \star T'))} \quad \text{(T-Fork)}$$

Figure 3.14: Syntax and typing of the fork operator

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 a new thread. 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 the signature of **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[20] chose an implementation with one argument, however with a similar requirement of a metalanguage of parallel terms $e \,|\, e'$.

I chose to use a primitive 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 $e \,|\, e'$. A primitive with several arguments can be elegantly simulated by composing several binary primitives.

To stay within the monad I decided to have the arguments as already boxed terms, that is of type **con** T and **con** T' for some T and T'. The **fork** primitive is not curried, however this behaviour can be simulated by:

$$\lambda x : \mathbf{con}\ T.(\lambda y : \mathbf{con}\ T'.(\mathbf{fork}\ x\ y))$$

For simplicity, I describe **fork** as the curried higher-order function: given the first argument it returns a function that takes only one parameter and will then behave as the parallel composition. This gives the following signature:

$$\mathbf{fork} : \mathbf{con}\ T \rightarrow (\mathbf{con}\ T' \rightarrow R) \tag{T-Fork1}$$

where R is the as yet undescribed return type. Currying allows partial application of **fork** and to be passed around as a value with only one edge filled.

To keep within the monad, I require 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, like the ones mentioned in Section 1.4, have varying return semantics. For example, **join** would return the pair of values resulting from the two expressions giving $R = \mathbf{con}\ (T \star T')$. Another primitive, **choose** would pick one and discard the other: $R = \mathbf{con}\ (T + T')$.

I wanted to provide a combination of these: 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 $R = \mathbf{con}\ ((T \star \mathbf{con}\ T') + ((\mathbf{con}\ T) \star T'))$ could describe this behaviour. The full signature is shown in (Fork-Full-Signature).

At first glance, this seems to be a rather complex signature but it is versatile enough to implement both **join** and **choose** 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 possibly non-interleaving semantics in mind. However, that led to a blow up in the number of rules, the complexity of the signature of **fork** ($R = \textbf{con}\ ((T \star \textbf{con}\ T') + ((\textbf{con}\ T) \star T') + (T \star T'))$) 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

$$\textbf{fork} : \underbrace{\textbf{con}\ T}_{\text{left computation}} \to \overbrace{\textbf{con}\ T'}^{\text{right computation}} \to \textbf{con}\ (\underbrace{(\overbrace{T}^{\text{finished value}} \star \textbf{con}\ T')}_{\text{left edge finished}} + \underbrace{((\textbf{con}\ \overbrace{T}^{\text{partially reduced}}) \star T')}_{\text{right edge finished}}) \tag{Fork-Full-Signature}$$

This corresponds to the type shown in rule (T-Fork) in Figure 3.14.

The operational semantics of the MOCaml concurrency features

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

$$\textbf{comp}\ e \xrightarrow[s]{e} e \tag{R-DoComp}$$

Figure 3.15: Operational semantics of the computation placeholder

Figure 3.15 shows the reduction rule (R-DoComp) for a computation place holder. It only exists in this form in the logical model: the placeholder will be fully replaced by a call to an OCaml function that returns an expression. A simple example could be Listing 3.3. This computation placeholder is then reduced by the case in Listing 3.4. Computation placeholders serve as a tool to prove, in Section 4.1, that the sequence of actual code calls obey certain order properties.

```
E_comp (fun _ -> print_string "Hello!"; E_unit)
```

Listing 3.3: OCaml computation placeholder example

```
let rec xJO_red12 p1 p2 =
  match (p1, p2) with
  ...
  | (E_comp e, s) -> (e ())
  ...
```

Listing 3.4: Reduction case for computation placeholders in the runnable OCaml

I chose to always reduce a place holder to an expression within the framework to provide an easy way for flexible, dynamic behaviour. This also means that the typing statement in Figure 3.12 is not enforced: it is assumed that the user pays attention to the type of the returned expression. Future work could include a way to type check expressions and to modify the extracted OCaml to enforce this typing rule.

Values

v ::= $\ldots$ *values:*
live e *live*

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

$$\textbf{ret}\ \ v \xrightarrow[s]{\tau} \textbf{live}\ v \qquad \text{(R-Return)}$$

$$\frac{e \xrightarrow[s]{rl} e'}{\textbf{ret}\ \ e \xrightarrow[s]{rl} \textbf{ret}\ \ e'} \qquad \text{(R-Evalret)}$$

$$\textbf{live}\ v \gg= e' \xrightarrow[s]{\tau} e'\ v \qquad \text{(R-Dobind)}$$

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

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

Figure 3.16: Operational semantics of the monadic primitives: ret and bind

Figure 3.16 shows the operational semantics of the concurrent type constructor **ret** and the operation **bind**. The **ret** operation first reduces its argument to a value with rule (R-Evalret) and then boxes it up silently as a concurrent value with rule (R-Return).

Bind evaluates left to right: it evalutes the expression on the left to a concurrent box with rule (R-Evalbind), reduces the internal expression of the box with rule (R-Movebind) and then silently passes it to the expression on the right with rule (R-Dobind). This fits well with the idea of sequencing operations: any side effect e' might have happens after the side effects of e.

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

$$\frac{e \xrightarrow[s]{el} e'' \qquad e' \notin \text{Values}}{\textbf{fork } (\textbf{live } e)(\textbf{live } e') \xrightarrow[L\ s]{rl} \textbf{fork } (\textbf{live } e'')(\textbf{live } e')} \quad \text{(R-Forkmove1)}$$

$$\frac{e' \xrightarrow[s]{rl} e'' \qquad e \notin \text{Values}}{\textbf{fork } (\textbf{live } e)(\textbf{live } e') \xrightarrow[R\ s]{rl} \textbf{fork } (\textbf{live } e)(\textbf{live } e'')} \quad \text{(R-Forkmove2)}$$

$$\textbf{fork } (\textbf{live } v)(\textbf{live } e') \xrightarrow[s]{\tau} \textbf{live } \textbf{left}\{v, (\textbf{live } e')\} \quad \text{(R-Forkdeath1)}$$

$$\textbf{fork } (\textbf{live } e)(\textbf{live } v') \xrightarrow[s]{\tau} \textbf{live } \textbf{right}\{(\textbf{live } e), v'\} \quad \text{(R-Forkdeath2)}$$

$$\frac{e \xrightarrow[s]{rl} e''}{\textbf{fork } \ e\ e' \xrightarrow[s]{rl} \textbf{fork } \ e''\ e'} \quad \text{(R-Forkeval1)}$$

$$\frac{e' \xrightarrow[s]{rl} e''}{\textbf{fork } \ v\ e' \xrightarrow[s]{rl} \textbf{fork } \ v\ e''} \quad \text{(R-Forkeval2)}$$

Figure 3.17: Operational semantics of the fork operator

In Figure 3.17 I give the detailed operational semantics of **fork**. First, **fork** reduces its arguments, left-to-right, to **live** boxes by rule (R-Forkeval1) and then by rule (R-Forkeval2). This is the first time the selection operator of the reduction relation is explicitly specified. If neither edge is a value, decision L reduces the left edge with rule (R-Forkmove1) while a decision R reduces the right edge with rule (R-Forkmove2). When either edge contains a value, **fork** is forced to silently reduce to the respective tagged value with rules (R-Forkdeath1) and (R-Forkdeath2), regardless of the decision. This gives a finer grade of control for the user to choose what to do with the partially reduced edge, however it increases overhead as value checks always have to be carried out on both edges.

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

There are four types of objects in the Coq semantics of MOCaml:

1. **Inductive sets** are the inductive data structures of Coq. These were translated from expression grammar objects. As an example, type expressions were translated from the Ott version in Listing 3.5 to the Coq equivalent in Listing 3.6.

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

Listing 3.5: Ott type expressions

```
Inductive typexpr : Set :=
 | TE_unit : typexpr
 | TE_arrow (t:typexpr) (t':typexpr)
 | TE_prod (t:typexpr) (t':typexpr)
 | TE_concurrent (t:typexpr)
 | TE_sum (t:typexpr) (t':typexpr).
```

Listing 3.6: Coq type expression

Grammar terms have a one-to-one correspondence except for (t) that has no counterpart: the flag S tells Ott that it is only syntactic sugar. Syntactic sugar, as explained in Section 2.4.1, does not get translated to a type constructor, but uses the instruction in the homomorphism:

```
{{ ich [[t]] }}
```

2. **Fixpoints**, the functions of Coq, 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 auxiliary functions

used in these definitions. The value subgrammar shown in Listing 3.7 was translated to the fixpoint in Listing 3.8.

```
value, v :: V_ ::=
  | unit                                    ::   :: unit
  | function value_name : t -> e            ::   :: function
  | live e                                  ::   :: live_expr
  | inl v                                   ::   :: taggedleft
  | inr v                                   ::   :: taggedright
  | { v , v' }                              ::   :: valuepair
  | ( v )                                   :: S :: paren
     {{ ich [[v]] }} {{ ocaml [[v]] }}
```

Listing 3.7: Ott value subgrammar

```
Fixpoint is_value_of_expr (e:expr) : Prop :=
  match e with
  | E_ident value_name5 ⇒ False
  | E_unit ⇒ True
  | E_apply e' e'' ⇒ False
  | E_bind e' e'' ⇒ False
  | E_function x t e' ⇒ True
  | E_fix e' ⇒ False
  | E_comp e' ⇒ False
  | E_live_expr e' ⇒ True
  | E_pair e' e'' ⇒ (is_value_of_expr e' ∧ is_value_of_expr e
    '')
  | E_proj1 e' ⇒ False
  | E_proj2 e' ⇒ False
  | E_fork e' e'' ⇒ False
  | E_ret e' ⇒ False
  | E_taggingleft e' ⇒ is_value_of_expr e'
  | E_taggingright e' ⇒ is_value_of_expr e'
  | E_case e' x1 e1 x2 e2 ⇒ False
end.
```

Listing 3.8: Coq value subgrammar

Ott offers single and multiple substitution predicates for its destination languages. These are implemented as fixpoints in Coq. Expression substitution(Listing 3.9) is an example of these automatically generated fixpoints.

```
Fixpoint subst_expr (e:expr) (x:value_name) (e':expr)
 {struct e'} : expr :=
  match e' with
   | E_ident y ⇒ if eq_value_name y x then e else (E_ident y)
   | E_unit ⇒ E_unit
   | E_apply e'' e''' ⇒ E_apply (subst_expr e x e'')
                                (subst_expr e x e''')
   ...
end
```

Listing 3.9: Coq expression substitution

3. **Logical inductive sets** are inductively defined sets of propositions. The reduction relation and the typing relation is represented as logical inductive set. For example a clause in the reduction relation was translated from the simple inference rule presentation in Listing 3.10 to a proposition in the logical inductive set JO_red in Listing 3.11.

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

Listing 3.10: Ott reduction relation example

```
Inductive JO_red : expr → select → redlabel → expr → Prop := (* defn red *)
 | JO_red_context_app1 : ∀ (e e':expr) (s:select) (rl:redlabel) (e'':expr),
     JO_red e s rl e'' →
     JO_red (E_apply e e') s rl (E_apply e'' e')
  ...
```

Listing 3.11: Coq reduction relation example

4. A **lemma** was also generated about the equality of variables. This lemma is shown in Listing 3.12.

```
Lemma eq_value_name: ∀ (x y : value_name), {x = y} + {x <> y}.
Proof.
  decide equality; auto with ott_coq_equality arith.
Defined.
```

Listing 3.12: Coq variable equality lemma

3.2.2 Extractable reduction relation

As I mentioned in Section 2.4.2, Coq provides built-in extraction facilities to OCaml and Haskell, but not from inductive sets and fixpoints that involve propositions, more specifically the *Prop* sort. However, with the extraction plugin developed by Delahaye et al[16, 44] I can extract such programs.

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 could extract the original reduction relation generated by Ott.

The first 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 a fatal error (`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(Listing B.1). Listings 3.13 and 3.14 are a good example of how this transformation happens.

```
| JO_red_evalbind : ∀ (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 3.13: Coq reduction clause with unsafe assumption

```
| XJO_red_evalbind : ∀ (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 3.14: Coq extractable reduction clause with safe assumption

The second issue was due to the experimental nature of the plugin: the optimisations and inference algorithm ran out of stack space when invoked with all 31 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(in Listing 3.6) are extracted as tagged variants or type constructors.
- Fixpoints(like Listing 3.8) and logical inductive sets like the expression substitution(in Listing 3.9) and xJO_red12, the single step reduction relation are extracted as functions.
- The metavariable `value_name` is an OCaml int instead of a constructor based extraction 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.

3.3.2 Hand modifications and justifications

I have made a number of modifications to the OCaml code to make it runnable. The Coq extraction defines the computation place holders as E_comp of expr. I changed expr to be an OCaml function unit -> expr to include external computations and inserted the call to these functions at the relevant places. An example could be Listing 3.15, which, when reduced, prints "Hello!" and behaves as the unit value from then on.

```
E_comp (fun _ -> print_string "Hello!"; E_unit)
```

Listing 3.15: OCaml computation placeholder example

Furthermore I have changed the way `select` is implemented. The extraction results in the type in Listing 3.16 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 3.17.

```
type select = __select Lazy.t
and __select =
| Seq of decision * select
```

Listing 3.16: OCaml lazy select

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

Listing 3.17: OCaml stream select

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 and Coq pattern matching differs. In Coq logical inductive definition cases act as a large disjunction, instead of a sequential tried: if an earlier case fails in OCaml the match fails, while Coq would just keep going.

Tollitte[44] made progress on the merging of cases when one case subsumes an other. However, in the case of MOCaml, these were not always correctly identified. For example the case in Lines 7 to 10 in Listing 3.19 subsumes the case in Listing 3.18. 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 arguments. This is an issue as any function applied to an argument that can be reduced will fail, as the case in Listing 3.18 comes first.

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

Listing 3.18: 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 3.19: OCaml original application case

My solution was to insert the correct step into the failing match as in Listing 3.20. Note, that taking that reduction may fail, but that is the expected behaviour.

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

Listing 3.20: 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 3.19 is an example of an unnecessary default case: 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. I simply removed the superfluous 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 3.21 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 3.21: 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

The multiple layers of automatic naming make development in raw MOCaml cumbersome. To make things easier for the user 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 e,
  ```

  ```
  let boxc f = E_live_expr (E_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 = E_fork (a, b)
  ```

- Round-robin and random schedulers in Listing 3.22 and Listing 3.23 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 3.22: 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 3.23: OCaml random scheduler

Chapter 4

Evaluation

In this chapter I evaluate the project from two perspectives: I explore and prove some theoretical properties of the semantics and I evaluate the performance of MOCaml against other concurrency implementations.

4.1 Theoretical evaluation

A monadic concurrency framework should satisfy a number of properties: monadic laws, process calculus axioms. These requirements are all phrased as equivalence between expressions 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 should 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 results and side-effects. Results are the return values of an expression, side-effects are the computation placeholders invoked by the system. That is why I phrased the operational semantics with labelled transitions: the labels are descriptions of the side-effects.

Labelled transition systems, like the operational semantics I have defined, are a common way to describe systems with side-effects. As the selection operator is external and not determined by the system I consider it and therefore the transition system non-deterministic. Brookes[13] surveys various equivalence relations over non-deterministic labelled transitions systems. I chose to follow Milner[33] and used weak bisimilarity.

4.1.1 Weak bisimilarity

Bisimilarity is a standard method for establishing behavioural equivalence of a non-deterministic language used by Milner for CCS[33], Pierce and Sangiorgi for π-calculus[36] and Bergstra for the Algebra of Communicating Processes[10, 9] among others.

In the following definitions I use p, q, e (and p', q', e' etc.) as expressions, α as an atomic action, τ as a silent action, σ as an action that may be either silent or atomic and $p \xrightarrow{\sigma} p'$ as the statement that for some selection value p may transition in a single step to p' with a side-effect σ.

Definition 4.1.1 (Bisimilarity)**.** Two expressions p and q are *bisimilar* if and only if

- For all q' such that $q \xrightarrow{\sigma} q'$ there is a p' such that $p \xrightarrow{\sigma} p'$ and p' and q' are bisimilar.
- For all p' such that $p \xrightarrow{\sigma} p'$ there is a q' such that $q \xrightarrow{\sigma} q'$ and p' and q' are bisimilar.

Weak bisimilarity is an extension to bisimilarity in which not all reductions are observable: Milner[33] introduces τ or silent reductions that do not have observable side-effects. In a setting with τ reductions it would be unnecessarily restrictive to require that two expressions simulate the unobservable behaviour of each other.

Definition 4.1.2 (Silent reduction)**.** *p silently reduces to p' or $p \overset{\tau}{\Rightarrow} p'$ if and only if $(p, p') \in \left(\xrightarrow{\tau}\right)^*$, that is, the two expressions are in the transitive, reflexive closure of $\xrightarrow{\tau}$. More simply if p can reduce to p' with zero or more τ steps.*

Definition 4.1.3 (Weak reduction)**.** *p weakly reduces to p' with side-effect σ or $p \overset{\sigma}{\Rightarrow} p'$ if and only if*

- *in the case $\sigma = \tau$, $p \overset{\tau}{\Rightarrow} p'$*
- *in the case $\sigma = \alpha$, there exist p_1, p_2 expressions such that $p \overset{\tau}{\Rightarrow} p_1$, $p_1 \xrightarrow{\alpha} p_2$, $p_2 \overset{\tau}{\Rightarrow} p'$*

A simplified version of Milner's definition for weak bisimilarity states:

Definition 4.1.4 (Milner's weak bisimulation)**.** A relation $\mathcal{R}$ between expressions is a *weak bisimulation* if and only if for all expressions p and q such that $(p, q) \in \mathcal{R}$ or $p \,\mathcal{R}\, q$:

- For all q' such that $q \overset{\sigma}{\Rightarrow} q'$ there is a p' such that $p \overset{\sigma}{\Rightarrow} p'$. and $(p', q') \in \mathcal{R}$.
- For all p' such that $p \overset{\sigma}{\Rightarrow} p'$ there is a q' such that $q \overset{\sigma}{\Rightarrow} q'$. and $(p', q') \in \mathcal{R}$.

Weak bisimulation has a number of useful properties: if $\mathcal{R}$ and $\mathcal{S}$ are weak bisimulations then $\mathcal{R}^{-1}, \mathcal{R}^{*}, \mathcal{R}^{+}, \mathcal{R} \cup \mathcal{S}$ and $\mathcal{R} \circ \mathcal{S}$ are all weak bisimulations.

Sangiorgi[38] remarks that there is an equivalent definition that uses only a single step:

Definition 4.1.5 (Sangiorgi's weak bisimulation)**.** A relation $\mathcal{R}$ between expressions is a *weak bisimulation* if and only if for all expressions p and q such that $(p, q) \in \mathcal{R}$ or $p \mathcal{R} q$:

- For all q' such that $q \xrightarrow{\sigma} q'$ there is a p' such that $p \overset{\sigma}{\Rightarrow} p'$. and $(p', q') \in \mathcal{R}$.
- For all p' such that $p \xrightarrow{\sigma} p'$ there is a q' such that $q \overset{\sigma}{\Rightarrow} q'$. and $(p', q') \in \mathcal{R}$.

I proved the equivalence of these two definitions for the transition system of MOCaml. This second definition is much easier to show in Coq as Coq generates an inversion lemma for a single step, as used in Sangiorgi's definition, however not for a weak reduction required by Milner's definition. This is a marked difference: the proofs in the following section took around five thousand lines of code in a previous iteration with Milner's definition, while only a few hundred lines with Sangiorgi's definition.

Definition 4.1.6 (Weak bisimilarity)**.** Two expressions p and q are *weakly bisimilar* or $p \approx q$ if there is a weak bisimulation $\mathcal{R}$ such that $(p, q) \in \mathcal{R}$.

Definition 4.1.6 is equivalent to saying that weak bisimilarity $\approx$ is the union of all weak bisimulations. Weak bisimilarity is an equivalence relation: it is reflexive, symmetric and transitive.

While weak bisimilarity perfectly captures side-effect behaviour it does not capture equality of return values: all values are weakly bisimilar as values do not reduce. Therefore in many cases I will use a restriction of weak bisimilarity:

Definition 4.1.7 (Value equal)**.** A weak bisimulation $\mathcal{R}$ is *value equal* if for all v, v' values if $v \mathcal{R} v'$ then $v = v'$

Definition 4.1.8 (Value equal weak bisimilarity)**.** Two expressions p and q are *value equal weakly bisimilar* or $p \approx_v q$ if there is a value equal weak bisimulation $\mathcal{R}$ such that $p \mathcal{R} q$.

These definitions also enjoy the same properties as their non-restricted counterparts. All of these definitions and some supporting lemmas can be found in `weakBisimulations.v`.

4.1.2 Properties of the logical model

In this section I introduce a number of properties that I required from the implementation and the way they apply. I give a slightly informal statement for each of these properties and note which proof file contains the formal proof. Appendix B gives the precise Coq statements of the theorems along with some supporting lemmas and structures.

First, I describe the three monadic laws mentioned in Section 2.3. Then I attempt to show some of the basic process algebra axioms used by Bergstra for his Algebra of Communicating Processes[8, 10] and Milner for his Calculus for Communicating Systems[33]. These properties can be summarized as follows:

$$\begin{aligned} x|y &= y|x && \text{(| commutativity)} \\ (x|y)|z &= x|(y|z) && \text{(| associativity)} \\ (x \cdot y) \cdot z &= x \cdot (y \cdot z) && \text{($\cdot$ associativity)} \\ \delta|x &= x && \text{(| deadlock)} \\ \delta \cdot x &= \delta && \text{($\cdot$ deadlock)} \end{aligned}$$

Sequential composition or bind is denoted by $\cdot$ and a process that cannot move forward is denoted by δ. Bergstra[8] calls process δ *deadlocked.* In the above axioms | denotes "parallel composition" which resembles the **fork** operator of MOCaml.

Monadic laws

The formal proofs of the following theorems about the monadic laws governing **ret** and bind $\gg=$ can be found in `mconbaseMonProofs.v`.

Theorem 4.1.9 (Return is left neutral to bind)**.** *For all expressions e, f, if f is a value then*

$$\textbf{\textit{ret}}\ e \gg= f \quad \approx_v \quad f\,e$$

The left neutrality has a caveat: it is only true if the function is a value. If this was not the case then the left hand side of the weak bisimilarity would first reduce f and then e. This was a conscious decision : it felt unintuitive to reduce the later part of a sequencing first.

Theorem 4.1.10 (Return is right neutral to bind)**.** *For any value v*

$$\textbf{\textit{live}}\ v \gg= \textbf{\textit{ret}} \quad \approx_v \quad \textbf{\textit{live}}\ v$$

Theorem 4.1.11 (Return is right neutral to bind, extended)**.** *For any expression e*

$$\textbf{\textit{live}}\ e \gg\!= \textbf{\textit{ret}} \quad \approx_v \quad \textbf{\textit{ret}}\ e$$

The general statement of the right neutrality is

$$e \gg\!= \textbf{ret} \quad \approx_v \quad e$$

However, this is not true in general for MOCaml. If e reduces to a live expression that has an unreduced expression boxed then the right hand side does not generate the side-effects of the boxed up computation. However, it is possible to show that if e always reduces to a boxed up value then right neutrality holds.

Theorem 4.1.12 (Bind is associative)**.** *For any expression e, values f, g and variable x, if x appears neither in f nor in g either bound or free*

$$(e \gg\!= f) \gg\!= g \quad \approx_v \quad e \gg\!= (\lambda x.(f\,x) \gg\!= g)$$

This corresponds to Equation ($\cdot$ associativity) in Bergstra's axioms. The caveat mentioned about non-capture avoiding substitution in Section 3.1.1 appears in the associativity statement as well: x needs to be a fresh variable to avoid substitution to unintended variables.

Fork commutativity

The proofs of **fork** commutativity and associativity may be found in `forkProofs.v`.

Theorem 4.1.13 (Fork commutative weak bisimilarity)**.** *For all expressions e, e'*

$$\textbf{\textit{fork}}\,(\textbf{\textit{live}}\ e)\,(\textbf{\textit{live}}\ e') \quad \approx \quad \textbf{\textit{fork}}\,(\textbf{\textit{live}}\ e')\,(\textbf{\textit{live}}\ e)$$

Showing the weak bisimilarity of **fork** and the commuted version using Sangiorgi's definition of weak bisimilarity is very simple: for every step there is a corresponding single step of the respective expression in the commuted version. However the return values of the simply commuted **fork** are not equivalent to those of the original. I solved this by sequencing a function that "swaps" the result:

Theorem 4.1.14 (Fork commutative value equal weak bisimilarity)**.** *For all expressions e, e'*

$$\textbf{\textit{fork}}\,(\textbf{\textit{live}}\ e)\,(\textbf{\textit{live}}\ e') \quad \approx_v \quad (\textbf{\textit{fork}}\,(\textbf{\textit{live}}\ e')\,(\textbf{\textit{live}}\ e)) \gg\!= \mathit{swapf}$$

```
Definition swapf : expr :=
    λ x : T. case x of left  x1 ⇒
                        ret(right {proj2 x1, proj1 x1})
                      | right x2 ⇒
                        ret(left  {proj2 x2, proj1 x2})
```

Listing 4.1: The *swapf* operator with λ notation

Listing 4.1 shows the definition of the *swapf* operator in a simplified λ notation. Listing B.7 shows the original Coq definition. The *swapf* operator just swaps the output of the commuted **fork** to the correct return value. Notice that *swapf* contains a type variable T. This variable is due to *swapf* being a family of operations, one for each **fork** return type. By binding *swapf* onto the commuted **fork** we get a value equal weak bisimilarity as *swapf* is always silent when applied to the return value of a **fork**.

Fork associativity

Fork in this implementation is not associative:

Theorem 4.1.15 (Fork is not associative)**.** *There are expressions a, b, c such that*

$$\boldsymbol{fork}\,(\boldsymbol{live}\ (\boldsymbol{fork}\,(\boldsymbol{live}\ a)\,(\boldsymbol{live}\ b)))\,(\boldsymbol{live}\ c) \quad \not\approx \quad \boldsymbol{fork}\,(\boldsymbol{live}\ a)\,(\boldsymbol{live}\ (\boldsymbol{fork}\,(\boldsymbol{live}\ b)\,(\boldsymbol{live}\ c))$$

The example is:

$$\textbf{fork}\ (\textbf{live}\ \textbf{fork}\ (\textbf{live}\ \textbf{unit})(\textbf{live}\ \textbf{unit}))(\textbf{live}\ \textbf{comp}\ e) \qquad \text{(R-Assoc)}$$

versus

$$\textbf{fork}\ (\textbf{live}\ \textbf{unit})\ (\textbf{live}\ \textbf{fork}\ ((\textbf{live}\ \textbf{unit})(\textbf{live}\ \textbf{comp}\ e))) \qquad \text{(L-Assoc)}$$

While R-Assoc may run the computation on the right, L-Assoc can only return immediately as the left edge is a value. In the proof for this I used a computation that returns **unit**.

Deadlock properties

The properties involving deadlock may be found in the file `forkDeadlock.v`.

Definition 4.1.16 (Deadlock)**.** An expression δ is *deadlocked* if it can reduce and any weak reduction it may make is silent and finishes in another *deadlocked* expression.

This definition is closer to the common understanding of livelock, however as Bergstra[8] explains: the "action" taken by δ is virtual, just acknowledging stagnation. Deadlock can occur by the incorrect use of the fixpoint combinator for example. In the proofs for the theorems below I used a stricter definition by only allowing deadlocked expressions that deterministically τ-step to themselves. An example of an expression satisfying this stricter definition is the term used to show that all types are occupied in (T-All).

Theorem 4.1.17 (Fork deadlock)**.** *For all expressions e, δ if δ is a deadlocked expression*

$$\boldsymbol{fork}\,(\boldsymbol{live}\ \delta)\,(\boldsymbol{live}\ e) \quad \approx \quad e$$

While the **fork** with a deadlocked edge is weakly bisimilar to the other edge, it has to be slightly modified to give value equality:

Theorem 4.1.18 (Fork deadlock value equivalence)**.** *For all expressions e, δ if δ is a deadlocked expression*

$$(\boldsymbol{fork}\,(\boldsymbol{live}\ \delta)\,(\boldsymbol{live}\ e)) \gg= \mathit{takeright} \quad \approx_v \quad \boldsymbol{ret}\ e$$

```
Definition takeright : expr :=
    λ x : T. case x of
              left  x1 ⇒ proj2 x1
            | right x2 ⇒ ret (proj1 x2).
```

Listing 4.2: The *takeright* function in λ notation

Listing 4.2 shows the definition of the *takeright* function in a simplified notation. Listing B.9 shows the original Coq definition. Much like *swapf*, *takeright* is also a family of functions based on the return type T of the preceding **fork**.

Theorem 4.1.19 (Bind deadlock)**.** *For all expressions e, δ if δ is a deadlocked expression*

$$\delta \gg= e \quad \approx_v \quad \delta$$

Note, however, the value equality comes for free: δ may never be a value.

Remarks on congruence

The various equivalences of Section 4.1.2 give meaningful insight into the behaviour of the mentioned expressions when considered in isolation. However, it falls short of the original aim of provide an equality that implies the ability to replace these expressions when composed with others.

This broader notion of equality is called a *congruence*. Intuitively an equivalence $\equiv$ between expressions is a congruence if it is compatible with the syntax and semantics of the language. That is, given an occurrence of an expression e in some context C (written as $C[e]$) and an equivalence $e \equiv e'$, e can be freely substituted: $C[e] \equiv C[e']$. In other words, a congruence is compositional.

Construction of a congruence relation over MOCaml is desirable, however not at all trivial. It is easy to see that the previously defined value equal weak bisimilarity (Definition 4.1.8) is not a congruence: λ-abstractions are only value equal weak bisimilar to themselves, as they are values. Even though the body may be substituted, the λ-abstraction would not be equivalent under this relation.

$$\lambda x : T.(\delta \gg= x) \quad \not\approx_v \quad \lambda x : T.\delta$$

However, weak bisimilarity also does not suffice: all values are trivially weakly bisimilar, since they do not reduce, but when substituted into a context arbitrary values will not behave the same. I decided to exclude proofs about congruences from the scope of this project. However, Howe's method[21], as described by Pitts[37], is a standard way of establishing a congruence relation. This method is based on finding a pre-congruence candidate relation $\lesssim$ and forming a congruence by Howe's construction. In simple languages it would follow that the symmetrisation of the pre-congruence relation $\lesssim \cap \lesssim^{\mathrm{op}} = \simeq$ is a congruence. However, the semantics of MOCaml are not deterministic. Pitts notes that in this case the congruence $\simeq$ is not equivalent to the symmetrisation of the pre-congruence relation. He suggests a method to overcome this, by using the transitive closure of the Howe's construction over an extension of applicative bisimilarity to open terms. However, in a setting with τ reductions this bisimilarity would have to be weakened which may cause further complications. I left the exploration of this method as potential further work.

4.1.3 Equivalence of the logical model and the extractable model

In this section I will prove the equivalence of the original logical model of the reduction relation generated by Ott and the relation extracted to OCaml.

Label erasure

As labels are only used to show properties of the system I decided not to include them in the extractable semantics: If labels were included, they would be completely ignored by the framework, but potentially introduce further runtime overhead.

To show that erasing labels is safe it suffices to show that there exists a deterministic partial function $(e, s, e') \to rl$ from reduction triples (start expression, selection operator and destination expression) to labels. First, I proved that for any given reduction triple if a label exists it is unique:

Theorem 4.1.20 (Unique labels)**.** *For all expressions e, e', selection value s and reduction labels rl, rl': if $e \xrightarrow[s]{rl} e'$ and $e \xrightarrow[s]{rl'} e'$ then $rl = rl'$.*

Equivalence

I denote the logical model reduction as $e \xrightarrow[s]{rl} e'$, as usual, while I denote the extractable reduction as $e \underset{s}{\dashrightarrow} e'$.

Theorem 4.1.21 (Model reduction implies extractable reduction)**.** *For all expressions e, e', selection s and reduction label rl, if $e \xrightarrow[s]{rl} e'$ then $e \underset{s}{\dashrightarrow} e'$.*

$$e \xrightarrow[s]{rl} e' \quad \Rightarrow \quad e \underset{s}{\dashrightarrow} e'$$

Theorem 4.1.21 means that if we denote the set of reduction triples (e, s, e') for the logical and extractable relations by S_L and S_X respectively, then $S_L \subseteq S_X$.

Theorem 4.1.22 (Extractable reduction implies model reduction)**.** *For all expressions e, e' and selection s if $e \underset{s}{\dashrightarrow} e'$ then there is a reduction label rl such that $e \xrightarrow[s]{rl} e'$.*

$$e \underset{s}{\dashrightarrow} e' \quad \Rightarrow \quad e \xrightarrow[s]{rl} e'$$

Theorem 4.1.22 means that $S_X \subseteq S_L$, therefore $S_X \equiv S_L \equiv S$ and furthermore by Theorem 4.1.20 it follows that there is a total function $L : S \to RL$ where RL is the set of reduction labels, as the reduction label is determined by the triple.

4.2 Performance

To evaluate the performance of the framework I followed Deleuze[17]. 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[17, p. 12-13], without verification. This is a much lighter implementation than the one in this project.

4. **promise**: A promise monad based implementation of lightweight concurrency[17, p. 13-15].

5. **tramp**: A lightweight implementation based on the trampolined style[17, p. 11-12].

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

7. **equeue**: An event-based programming style lightweight concurrency implementation[17, 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 for performance, based on the examples used in Kahn's process language paper[22]. I have implemented the examples in MOCaml using the random scheduler in Section 3.3.3 as the default scheduling algorithm. 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, which 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, and caml-shift version of August 2013.

My conjecture was that MOCaml will perform with similar characteristics as lightweight systems, but with a serious interpretive overhead.

4.2.2 Examples

The three examples provided by Deleuze are all process networks from Kahn[22]. 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 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. 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 $2^a3^b5^c$ 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.

Kpn

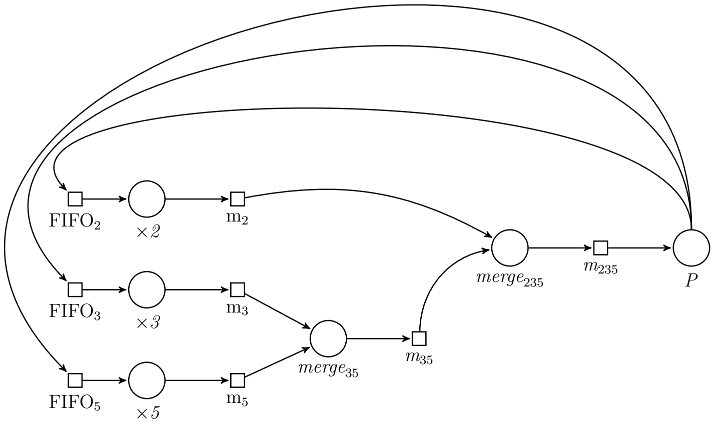


Figure 4.1: `kpn` process outline

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

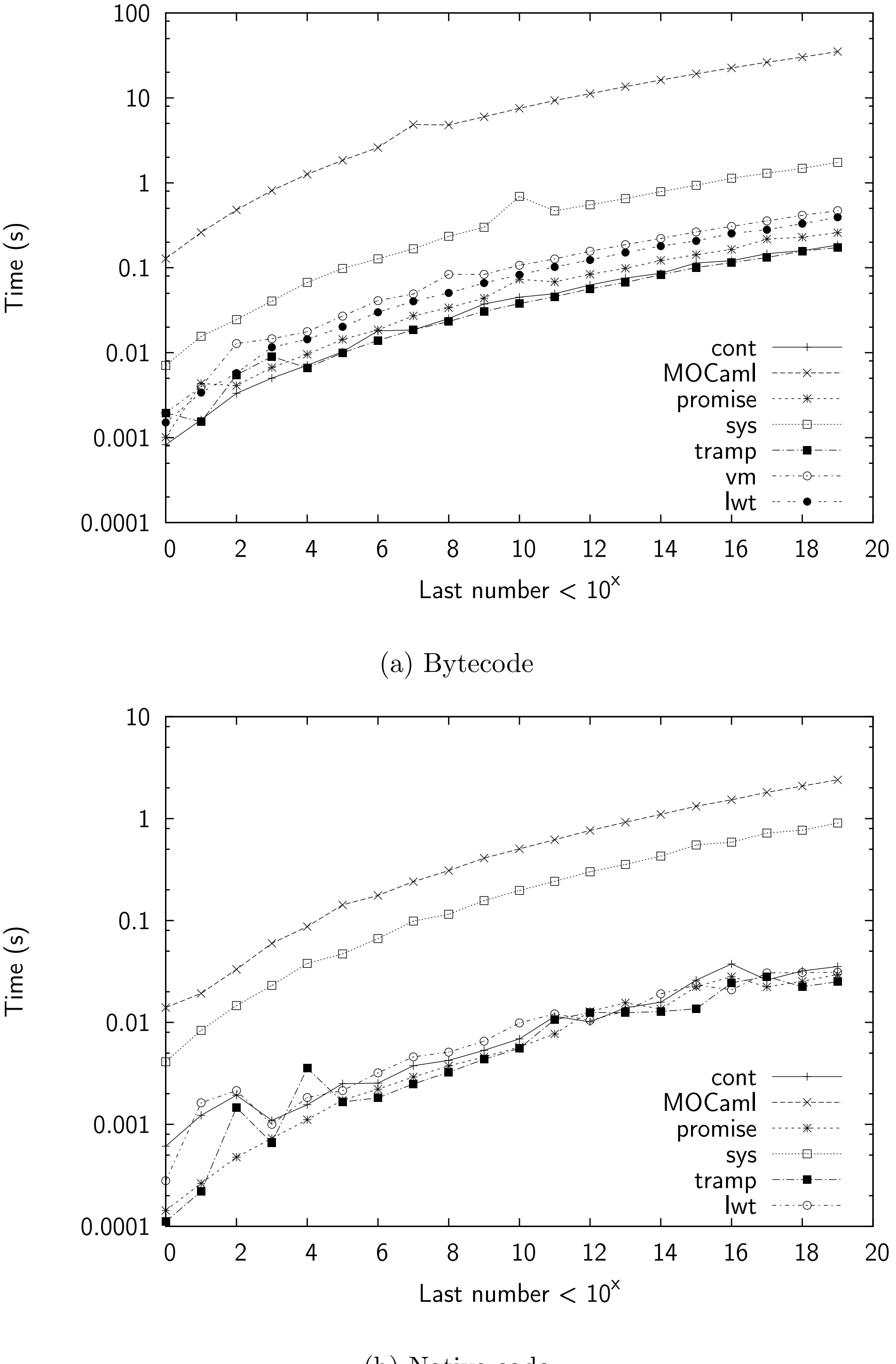
- The processes $\times 2$, $\times 3$, $\times 5$ multiply their input (taken from the corresponding FIFO) by 2, 3 and 5 respectively.
- The process $merge_{35}$ merges the output of $\times 3$, $\times 5$ processes using the corresponding message variables: it outputs the lower and fetches a new element that edge.

- Similarly, the process $merge_{235}$ merges the output of the $merge_{35}$ and $\times 2$ processes

- The process 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. As expected, Figure 4.2 shows that MOCaml exhibits exactly the same behaviour as other lightweight implementations, but with a serious overhead: it is 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.



(a) Bytecode

(b) Native code

Figure 4.2: `kpn` execution time

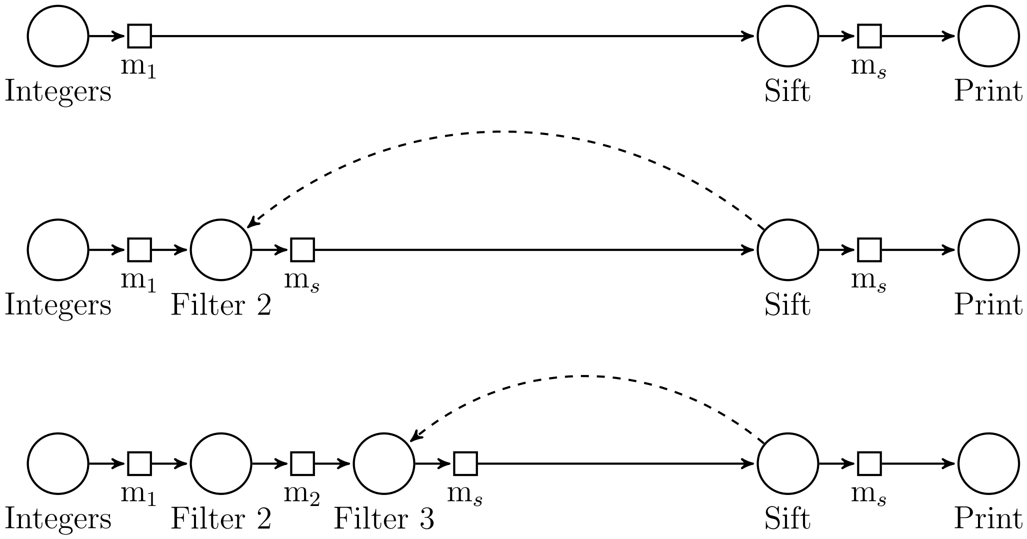


Figure 4.3: `sieve` process outline

Sieve of Eratosthenes

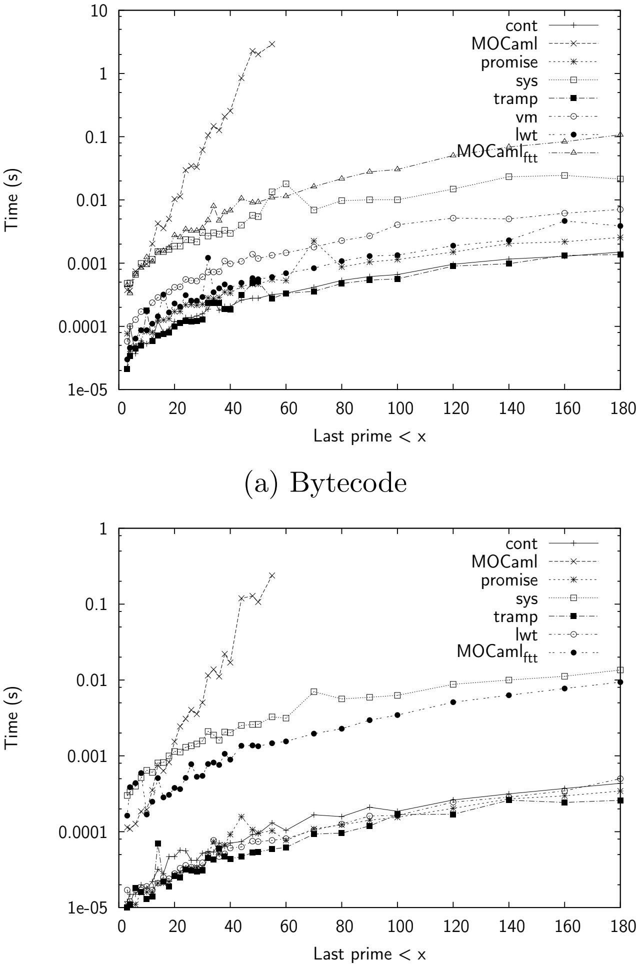
The sieve of Eratosthenes is one of the simplest algorithms to find primes:

1. Start with number 2 and consider all numbers greater than 2 unmarked.
2. Take the lowest unmarked number: it is a prime.
3. Mark all numbers divisible by the prime just found.
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. Sift also creates a new process, Filter p and inserts it between the input of Sift and Sift itself in the flow of numbers (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.

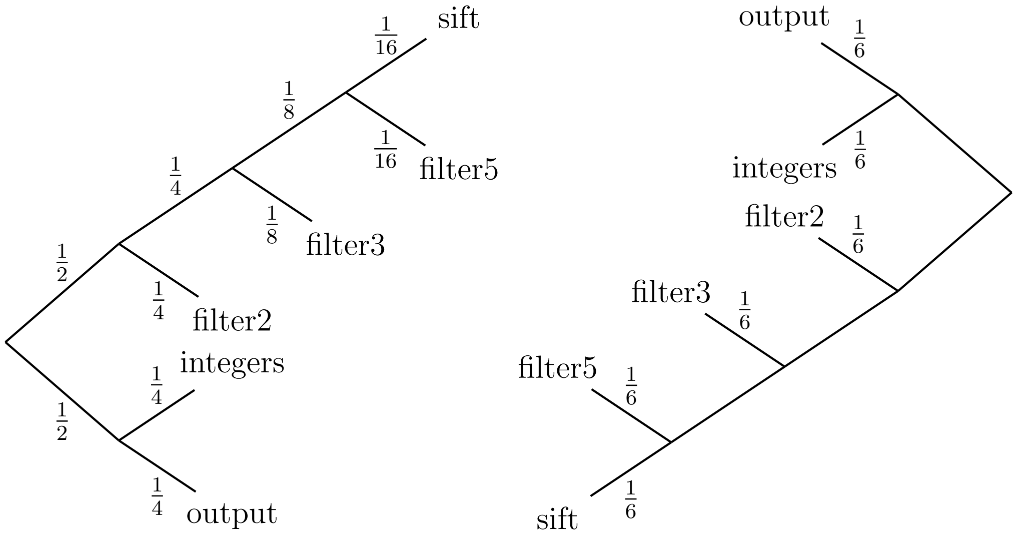


(a) Bytecode

(b) Native code

Figure 4.4: `sieve` execution time

As described in Section 3.3.3, the default MOCaml scheduler chooses threads to reduce at random .Unexpectedly, the default implementation (MOCaml) 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.



(a) Random without fork tree tracking

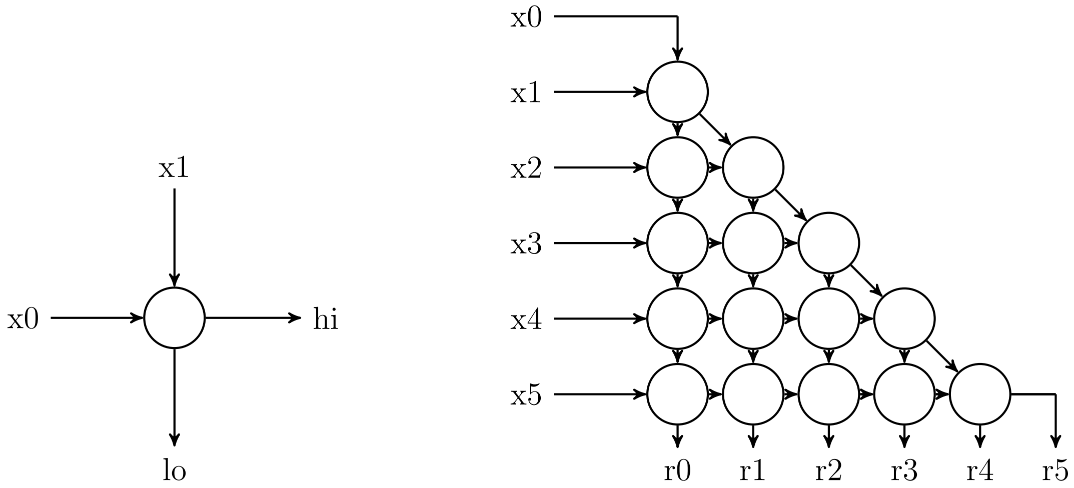
(b) Fork tree tracking

Figure 4.5: Sieve fork tree

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 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, although the default scheduler in the MOCaml concurrency framework is random, the user is not restricted to uniform probabilities for each decision. 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 MOCaml$_{\text{ftt}}$.

Concurrent sort



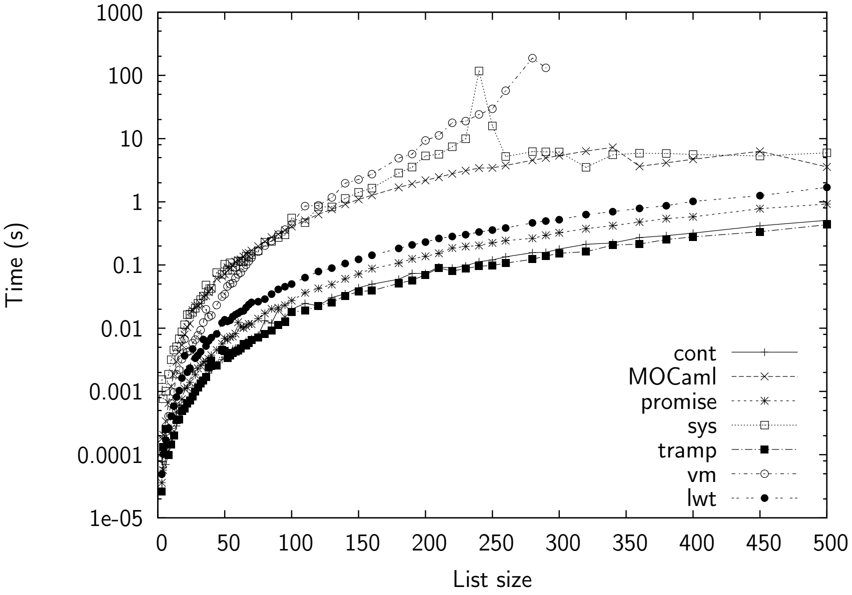
(a) Single comparator

(b) Full sorting network

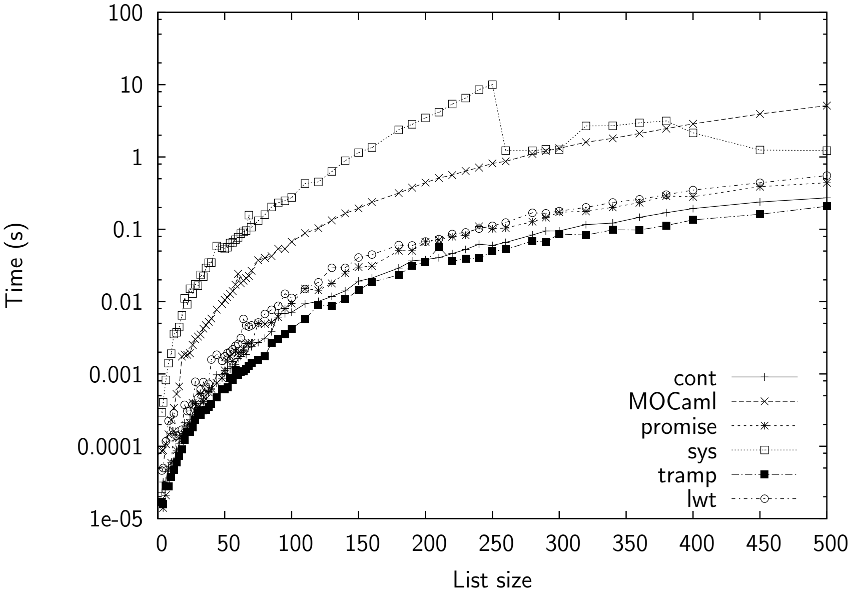
Figure 4.6: `sorter` process outline

The last example is concurrent sort. Sorting of a list of integers can be implemented by single comparator units which take in two values and output the lower on one edge and the higher on the other (Figure 4.6a). 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. Different schedulings of this network correspond to different algorithms, such as bubble sort and insertion sort. 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 performs very poorly, even though it is lighweight. On the other hand, the heavyweight **sys** implementation changes characteristics when the list grows larger than 250 elements and performs similarly to lightweight implementations. The memory use of my system is much higher than the other implementations (Figure 4.8).

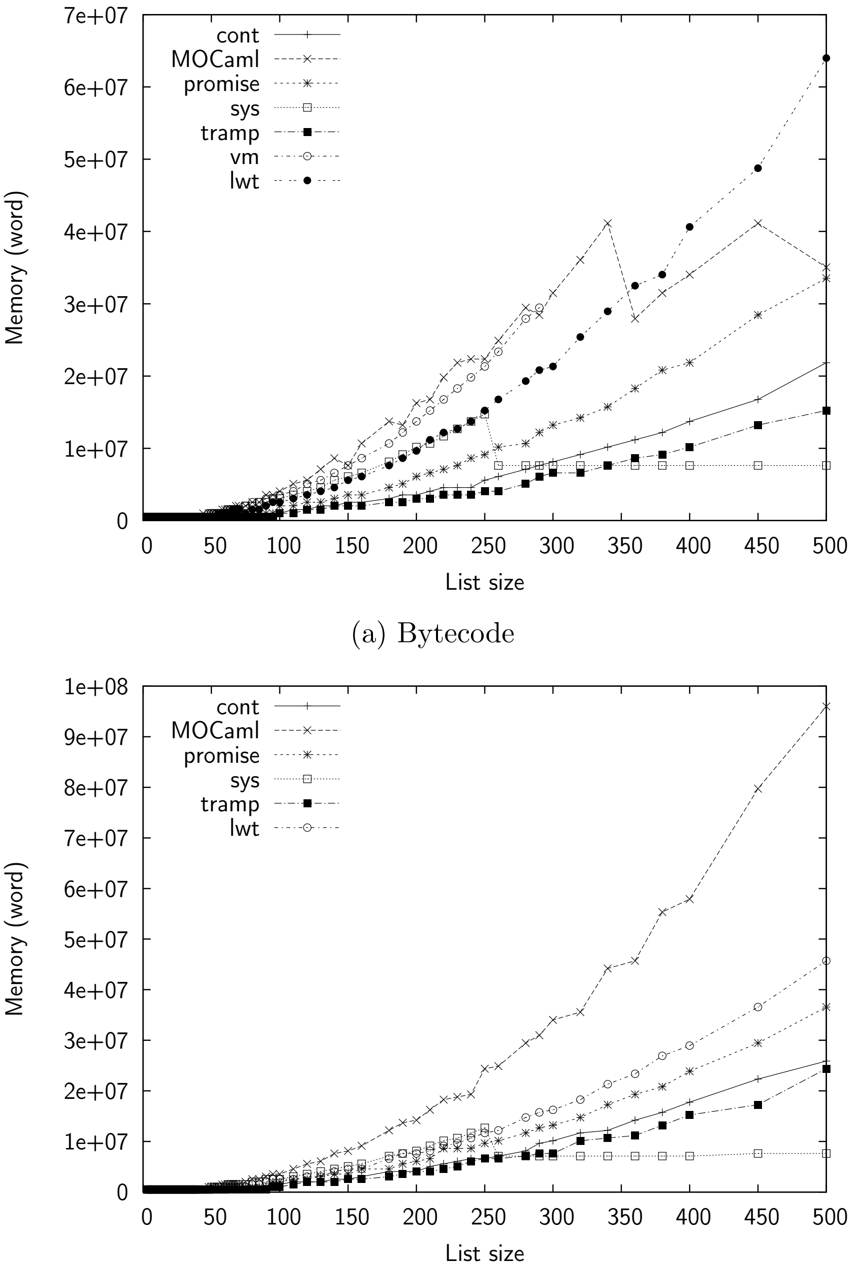


(a) Bytecode



(b) Native code

Figure 4.7: `sorter` execution time



(a) Bytecode

(b) Native code

Figure 4.8: `sorter` memory use

Chapter 5

Conclusion

In this project I have demonstrated that it is possible to build a lightweight concurrency framework in OCaml based on formal semantics. I defined the exact semantics of MOCaml in a simple format using Ott, providing precise, low level documentation of the framework. After translation to Coq, I proved a number of desirable properties of a concurrency library. These proofs act as high level, behavioural documentation of the system. I then successfully extracted OCaml code implementing the semantics. This code is supported by some syntactic sugar for construction of concurrency structures and scheduling to form a usable framework. I evaluated MOCaml against a number of other implementations based on performance to document the runtime characteristics of the framework. Overall, MOCaml met the expectations: its performance behaviour is the same as other lightweight concurrency frameworks but with a large overhead. This overhead is due to the trade-off between theoretical simplicity and efficiency.

The project left a number of theoretical questions intentionally open, including proofs of type preservation, progress and the nature of the congruence relation. On the practical side, MOCaml is only a minimal example of a concurrency framework: type checking, verified support for communication channels, locks, further concurrency primitives and handling of exceptions are all possible future extensions of this project.

More generally, I have demonstrated an efficient workflow for developing a library in a mainstream language underpinned by formal reasoning. The toolchain of Ott, Coq (extended with the logical inductive relation extraction plugin) and OCaml provides the necessary support to describe the intended semantics in an easy-to-read format, reason about said semantics and finally extract a program implementing it. An efficient workflow is necessary to make formal semantics based development a feasible option in software engineering.

There are many directions open for formal semantics driven development.

While an approach based on formal semantics is not warranted in all situations, foundational libraries and libraries with high safety requirements, like concurrency or security frameworks, can potentially benefit from the level of rigour used in this project. In these situations, the library is often developed first and documentation occurs later, based on semantics gleaned from the initial plan and some unit testing. This documentation is sometimes partial or has unverified assumptions, for example the unit testing did not cover all edge cases. Software built on an unverified library may have unexpected behaviour and hard to find, costly errors. This problem is largely eliminated by the approach taken in this project: developing the self-documenting semantics first and using verified tools to extract a program satisfying the specifications.

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] Stefan Berghofer. Program extraction in simply-typed higher order logic. In *Types for Proofs and Programs*, pages 21–38. Springer, 2003.

[8] Jan A Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and control*, 60(1):109–137, 1984.

[9] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theoretical computer science*, 37:77–121, 1985.

[10] Jan A Bergstra and Jan Willem Klop. Algebra of communicating processes. *Mathematics and Computer Science, CWI Monograph*, 1:89–138, 1986.

[11] 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.

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

[13] Stephen Brookes and William Rounds. Behavioural equivalence relations induced by programming logics. *Automata, Languages and Programming*, pages 97–108, 1983.

[14] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

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

[16] 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.

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

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

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

[20] 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.

[21] Douglas J Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

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

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

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

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

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

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

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

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

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

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

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

[33] Robin Milner. *A calculus of communicating systems.* Springer-Verlag New York, Inc., 1982.

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

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

[36] Benjamin C Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM (JACM)*, 47(3):531–584, 2000.

[37] AM Pitts. Howes method for higher-order languages. *Advanced Topics in Bisimulation and Coinduction*, 52:197–232, 2011.

[38] Davide Sangiorgi and Robin Milner. The problem of weak bisimulation up to. In *CONCUR'92*, pages 32–46. Springer, 1992.

[39] 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.

[40] 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/`.

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

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

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

[44] 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.

[45] 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

$$
\begin{array}{llll}
\mathit{value\_name},\ x & & & \\
\mathit{index},\ i,\ j,\ n,\ m & & & \\
\mathit{typexpr},\ t & ::= & & \\
 & | & \mathbf{tunit} & \\
 & | & \mathit{typexpr} \rightarrow \mathit{typexpr}' & \\
 & | & \mathit{typexpr} * \mathit{typexpr}' & \\
 & | & \mathbf{con}\, \mathit{typexpr} & \\
 & | & \mathit{typexpr} + \mathit{typexpr}' & \\
 & | & (\mathit{typexpr}) & \mathsf{S}
\end{array}
$$

$$
\begin{array}{llll}
\mathit{redlabel},\ \mathit{rl} & ::= & & \\
 & | & \tau & \\
 & | & \mathit{expr} &
\end{array}
$$

$$
\begin{array}{llll}
\mathit{expr},\ e & ::= & & \\
 & | & \mathit{value\_name} & \\
 & | & \mathbf{unit} & \\
 & | & \mathit{expr}\ \mathit{expr}' & \\
 & | & \mathit{expr} \gg= \mathit{expr}' & \\
 & | & \lambda\, \mathit{value\_name} : t.e & \mathsf{bind}\ \mathit{value\_name}\ \mathsf{in}\ e \\
 & | & \mathbf{fix}\, e & \\
 & | & \mathbf{comp}\, e & \\
 & | & \mathbf{live}\, e & \\
 & | & \{e, e'\} & \\
 & | & \mathbf{proj}_1 e & \\
 & | & \mathbf{proj}_2 e &
\end{array}
$$

	$\mid$	$\mathbf{fork}\, e\, e'$	
	$\mid$	$\mathbf{ret}\, e$	
	$\mid$	(expr)	S
	$\mid$	$\mathbf{left}\, e$	
	$\mid$	$\mathbf{right}\, e$	
	$\mid$	$\mathbf{case}\, e\, \mathbf{of}\, \mathbf{left}\, x_1 \Rightarrow e_1 \mid \mathbf{right}\, x_2 \Rightarrow e_2$	bind x_1 in e_1
			bind x_2 in e_2
	$\mid$	$\{v/x\}e$	M
	$\mid$	$\{\mathbf{fix}(\lambda x : t.e)/x'\}e$	M
$\mathit{value},\ v$	$::=$		
	$\mid$	$\mathbf{unit}$	
	$\mid$	$\lambda\, \mathit{value\_name} : \mathit{typexpr}.\mathit{expr}$	
	$\mid$	$\mathbf{live}\, e$	
	$\mid$	$\mathbf{left}\, v$	
	$\mid$	$\mathbf{right}\, v$	
	$\mid$	$\{v, v'\}$	
	$\mid$	(v)	S
Γ	$::=$		
	$\mid$	$\mathbf{empty}$	
	$\mid$	$\Gamma, \mathit{value\_name} : \mathit{typexpr}$	
$\mathit{formula}$	$::=$		
	$\mid$	$\mathit{judgement}$	
	$\mid$	$\mathbf{not}\, (\mathit{formula})$	
	$\mid$	$\mathit{value\_name} = \mathit{value\_name}'$	
	$\mid$	$\mathit{is\_value}e$	
	$\mid$	$e \notin \mathit{Values}$	
$\mathit{terminals}$	$::=$		
	$\mid$	$\rightarrow$	
	$\mid$	$.$	
	$\mid$	λ	
	$\mid$	$\vdash$	
	$\mid$	$\longrightarrow$	

$$\begin{array}{lll}
 & | & \{ \\
 & | & \} \\
 & | & [\\
 & | &] \\
 & | & \mathbf{con} \\
 & | & \mathbf{comp} \\
 & | & \mathit{exp} \\
 & | & \mathbf{live} \\
 & | & \mathbf{fix} \\
 & | & \mathbf{proj}_1 \\
 & | & \mathbf{proj}_2 \\
 & | & \tau \\
 & | & \gg= \\
 & | & * \\
 & | & , \\
 & | & \mathbf{left} \\
 & | & \mathbf{right} \\
 & | & \mathbf{case} \\
 & | & \mathbf{of} \\
 & | & \Rightarrow \\
 & | & + \\
 & | & : \\
 & | & \in
\end{array}$$

$$\begin{array}{lll}
\mathit{selectopt},\ o & ::= & \\
 & | & 1 \\
 & | & 2
\end{array}$$

$$\begin{array}{lll}
\mathit{select},\ s & ::= & \\
 & | & o\, s
\end{array}$$

$$\begin{array}{lll}
\mathit{Jtype} & ::= & \\
 & | & \mathit{value\_name} : \mathit{typexpr}\ \in\ \Gamma \\
 & | & \Gamma \vdash e : t
\end{array}$$

$$\begin{array}{lll}
\mathit{Jop} & ::= &
\end{array}$$

$$\begin{array}{lcl}
 & | & e \xrightarrow[s]{rl} e'
\end{array}$$

$$\begin{array}{lcl}
\mathit{judgement} & ::= & \\
 & | & \mathit{Jtype} \\
 & | & \mathit{Jop}
\end{array}$$

$$\begin{array}{lcl}
\mathit{user\_syntax} & ::= & \\
 & | & \mathit{value\_name} \\
 & | & \mathit{index} \\
 & | & \mathit{typexpr} \\
 & | & \mathit{redlabel} \\
 & | & \mathit{expr} \\
 & | & \mathit{value} \\
 & | & \Gamma \\
 & | & \mathit{formula} \\
 & | & \mathit{terminals} \\
 & | & \mathit{selectopt} \\
 & | & \mathit{select}
\end{array}$$

$\boxed{\mathit{value\_name} : \mathit{typexpr} \ \in \ \Gamma}$

$$\frac{}{\mathit{value\_name} : \mathit{typexpr} \ \in \ \Gamma, \mathit{value\_name} : \mathit{typexpr}} \quad \text{VTS}\textsc{in\_vn1}$$

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

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

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathbf{ret}\, e : \mathbf{con}\, t} \quad \textsc{Get\_ret}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : (\mathbf{con}\, t_1) \\ \Gamma \vdash e' : (\mathbf{con}\, t_2)\end{array}}{\Gamma \vdash \mathbf{fork}\, e\, e' : (\mathbf{con}\,((t_1 * (\mathbf{con}\, t_2)) + ((\mathbf{con}\, t_1) * t_2)))} \quad \textsc{Get\_fork}$$

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

$$\frac{\Gamma \vdash e : (t_1 * t_2)}{\Gamma \vdash \mathbf{proj}_1\, e : (t_1 * t_2) \to t_1} \quad \textsc{Get\_proj1}$$

$$\frac{\Gamma \vdash e : (t_1 * t_2)}{\Gamma \vdash \mathbf{proj}_2\, e : (t_1 * t_2) \to t_2} \quad \textsc{Get\_proj2}$$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t} \quad \textsc{Get\_value\_name}$$

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

$$\frac{\Gamma, x_1 : t_1 \vdash e : t}{\Gamma \vdash \lambda\, x_1 : t_1 . e : t_1 \to t} \quad \textsc{Get\_lambda}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathbf{live}\, e : \mathbf{con}\, t} \quad \textsc{Get\_live\_exp}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash (\mathbf{comp}\, e) : t} \quad \textsc{Get\_comp}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathbf{con}\, t\\ \Gamma \vdash e' : t \to \mathbf{con}\, t'\end{array}}{\Gamma \vdash e \gg\!= e' : \mathbf{con}\, t'} \quad \textsc{Get\_bind}$$

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

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

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

$$\frac{\begin{array}{l}\Gamma \vdash e : t + t'\\ \Gamma, x : t \vdash e' : t''\\ \Gamma, x' : t' \vdash e'' : t''\end{array}}{\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, \mathbf{left}\, x \Rightarrow e' |\, \mathbf{right}\, x' \Rightarrow e'' : t''} \quad \textsc{Get\_TCase}$$

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

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

$$\frac{}{\mathbf{comp}\, e \xrightarrow[s]{e} e} \quad \textsc{JO\_red\_docomp}$$

$$\frac{e \xrightarrow[s]{rl} e''}{\mathbf{fork}\, e\, e' \xrightarrow[s]{rl} \mathbf{fork}\, e''\, e'} \quad \textsc{JO\_red\_forkeval1}$$

$$\frac{e' \xrightarrow[s]{rl} e''}{\mathbf{fork}\; v\; e' \xrightarrow[s]{rl} \mathbf{fork}\; v\; e''} \quad \text{JO\_RED\_FORKEVAL2}$$

$$\frac{\begin{array}{c} e \xrightarrow[s]{rl} e'' \\ e' \notin \mathit{Values} \end{array}}{\mathbf{fork}\,(\mathbf{live}\; e)\,(\mathbf{live}\; e') \xrightarrow[1\,s]{rl} \mathbf{fork}\,(\mathbf{live}\; e'')\,(\mathbf{live}\; e')} \quad \text{JO\_RED\_FORKMOVE1}$$

$$\frac{\begin{array}{c} e' \xrightarrow[s]{rl} e'' \\ e \notin \mathit{Values} \end{array}}{\mathbf{fork}\,(\mathbf{live}\; e)\,(\mathbf{live}\; e') \xrightarrow[2\,s]{rl} \mathbf{fork}\,(\mathbf{live}\; e)\,(\mathbf{live}\; e'')} \quad \text{JO\_RED\_FORKMOVE2}$$

$$\frac{}{\mathbf{fork}\,(\mathbf{live}\; v)\,(\mathbf{live}\; e) \xrightarrow[s]{\tau} \mathbf{live}\,(\mathbf{left}\,((\{v, (\mathbf{live}\; e)\})))} \quad \text{JO\_RED\_FORKDEATH1}$$

$$\frac{}{\mathbf{fork}\,(\mathbf{live}\; e)\,(\mathbf{live}\; v') \xrightarrow[s]{\tau} \mathbf{live}\,(\mathbf{right}\,(\{(\mathbf{live}\; e), v'\}))} \quad \text{JO\_RED\_FORKDEATH2}$$

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

$$\frac{e \xrightarrow[s]{rl} e'}{\mathbf{ret}\; e \xrightarrow[s]{rl} \mathbf{ret}\; e'} \quad \text{JO\_RED\_EVALRET}$$

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

$$\frac{e \xrightarrow[s]{rl} e'}{(\mathbf{live}\; e) \gg= e'' \xrightarrow[s]{rl} (\mathbf{live}\; e') \gg= e''} \quad \text{JO\_RED\_MOVEBIND}$$

$$\frac{}{(\mathbf{live}\; v) \gg= e \xrightarrow[s]{\tau} e\; v} \quad \text{JO\_RED\_DOBIND}$$

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

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

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

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

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

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

$$\frac{}{\mathbf{proj}_1\{v, v'\} \xrightarrow[s]{\tau} v} \quad \text{JO\_RED\_PROJ1}$$

$$\frac{}{\mathbf{proj}_2\{v, v'\} \xrightarrow[s]{\tau} v'} \quad \text{JO\_RED\_PROJ2}$$

$$\frac{e \xrightarrow[s]{rl} e'}{\mathbf{proj}_1\, e \xrightarrow[s]{rl} \mathbf{proj}_1\, e'} \quad \text{JO\_RED\_PROJ1\_EVAL}$$

$$\frac{e \xrightarrow[s]{rl} e'}{\mathbf{proj}_2\, e \xrightarrow[s]{rl} \mathbf{proj}_2\, e'} \quad \text{JO\_RED\_PROJ2\_EVAL}$$

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

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

$$\frac{}{\mathbf{case}\,(\mathbf{left}\ v)\,\mathbf{of}\,\mathbf{left}\ x_1 \Rightarrow e_1 |\ \mathbf{right}\ x_2 \Rightarrow e_2 \xrightarrow[s]{\tau} \{v/x_1\}e_1} \quad \text{JO\_RED\_EVALCASEINL}$$

$$\frac{}{\mathbf{case}\,(\mathbf{right}\ v)\,\mathbf{of}\,\mathbf{left}\ x_1 \Rightarrow e |\ \mathbf{right}\ x_2 \Rightarrow e_2 \xrightarrow[s]{\tau} \{v/x_2\}e_2} \quad \text{JO\_RED\_EVALCASEINR}$$

$$\frac{e \xrightarrow[s]{rl} e'}{\mathbf{case}\ e\ \mathbf{of}\,\mathbf{left}\ x_1 \Rightarrow e_1 |\ \mathbf{right}\ x_2 \Rightarrow e_2 \xrightarrow[s]{rl} \mathbf{case}\ e'\ \mathbf{of}\,\mathbf{left}\ x_1 \Rightarrow e_1 |\ \mathbf{right}\ x_2 \Rightarrow e_2} \quad \text{JO\_RED\_EVALCASE}$$

Appendix B

Coq definitions and theorem statements

B.1 Supporting lemmas

```
Lemma red_not_value : ∀ (e e' : expr)(s : select) (rl : redlabel),
   e [ s ] ⟶ [ rl ] e' → (~(is_value_of_expr e)).
```

Listing B.1: If an expression reduces, it is not value

B.2 Weak bisimilarity definitions and theorems

```
Inductive tauStep : relation expr :=
 | tStep :  ∀ (e e' : expr) (s : select), JO_red e s RL_tau e' → tauStep e e'.

Definition tauRed : relation expr := (star tauStep).

Inductive totalDetTauStep : relation expr :=
 | ttStep : ∀ (e e' : expr),
   ( (tauStep e e') ∧
   (∀ (e'' : expr) (s : select) (l : label), JO_red e s (RL_labelled (l)) e'' → False) ∧
   (∀ (e''' : expr), tauStep e e''' → e' = e''')) → totalDetTauStep e e'.

Definition totalTauRed : relation expr := (star totalDetTauStep).

Inductive labRed : label → relation expr :=
 | lab_r : ∀ (e0 e1 e2 e3 : expr) (s : select) (l : label),
   tauRed e0 e1 ∧
   JO_red e1 s (RL_labelled(l)) e2 ∧
   tauRed e2 e3 → labRed l e0 e3.

Inductive weakred : redlabel → relation expr :=
 | weakred_T : ∀ ( e e' : expr ), tauRed e e' →
                weakred RL_tau e e'
 | weakred_L : ∀ ( e e' : expr) (l : label), labRed l e e' →
                weakred (RL_labelled l) e e'.
```

Listing B.2: Definitions of reduction relations

```
(* Milner *)
Definition isExprRelationWeakBisimilarity (R : relation expr) : Prop :=
   ∀ (p q : expr), R p q →
   ((∀ (p' : expr) (l : label),
     labRed l p p' →
      (exists (q' : expr), labRed l q q' ∧ R p' q' )) ∧
    (∀ (q' : expr) (l : label),
     labRed l q q' →
      (exists (p' : expr), labRed l p p' ∧ R p' q' )) ∧
    (∀ (p' : expr),
     tauRed p p' →
     (exists (q' : expr), tauRed q q' ∧ R p' q' )) ∧
    (∀ (q' : expr),
     tauRed q q' →
      (exists (p' : expr), tauRed p p' ∧ R p' q' ))
 ).

(* Sangiorgi *)
Definition isExprRelationStepWeakBisimilarity (R : relation expr) : Prop :=
   ∀ (p q : expr),
    R p q →
     ((∀ (p' : expr) (rl : redlabel) (s: select),
        p [ s ] ⟶ [ rl ] p' →
        (exists (q' : expr), weakred rl q q' ∧ R p' q' )) ∧
     (∀ (q' : expr) (rl : redlabel) (s : select),
        q [ s ] ⟶ [ rl ] q' →
        (exists (p' : expr), weakred rl p p' ∧ R p' q' ))).
```

Listing B.3: Definitions of weak bisimilarity

```
Definition WBSM : relation expr → Prop :=
  isExprRelationStepWeakBisimilarity

Lemma isExprRelationWeakBisimilarity_equiv_WBSM :
  ∀ (R : relation expr),
   isExprRelationWeakBisimilarity R ↔ WBSM R.

Lemma WBSM_comp : ∀ (R S : relation expr), WBSM R → WBSM S →
                    WBSM (comp R S).

Lemma WBSM_eeq : ∀ (R S : relation expr), eeq R S → WBSM R → WBSM S.

Lemma WBSM_trans : ∀ (R : relation expr), WBSM R → WBSM (trans R).

Lemma WBSM_star :∀ (R : relation expr), WBSM R → WBSM (star R).

Lemma WBSM_union2 : ∀ (R S : relation expr), WBSM R → WBSM S →
                      WBSM (union2 R S).
```

Listing B.4: Weak bisimilarity properties

B.3 Monadic laws

```
Inductive mon_left_id_rel : relation expr :=
| mon_left_id_bound : ∀ (e e' : expr), is_value_of_expr e' →
          mon_left_id_rel ((E_ret e) >>= e') (E_apply e' e)
| mon_left_id_inbound : ∀ (e e' : expr), is_value_of_expr e' →
          mon_left_id_rel ((E_live_expr e) >>= e') (E_apply e' e)
| mon_left_id_unbound : ∀ (e : expr), mon_left_id_rel (e) (e).

Theorem mon_left_id_wbsm : WBSM mon_left_id_rel.

Theorem mon_left_id_rel_vewbsm : VEWBSM mon_left_id_rel.

Inductive mon_right_id_rel : relation expr :=
| mon_right_id_bound : ∀ (a : expr) (x : value_name) (t : typexpr),
   is_value_of_expr a →
 mon_right_id_rel (E_live_expr (a) >>=
   (E_function x t (E_ret (E_ident x)))) (E_live_expr ( a))
| mon_right_id_inbound : ∀ (a : expr) (x : value_name) (t : typexpr)
   , is_value_of_expr a →
 mon_right_id_rel
   (E_apply (E_function x t (E_ret (E_ident x))) ( (a)))
   (E_live_expr ( a))
| mon_right_id_unbound : ∀ (a : expr), is_value_of_expr a →
 mon_right_id_rel  (E_ret a)  (E_live_expr a)
| mon_right_id_ret : ∀ (a : expr),
  mon_right_id_rel (E_live_expr a)  (E_live_expr a).

Theorem mon_right_id_wbsm : WBSM mon_right_id_rel.

Theorem mon_right_id_rel_vewbsm : VEWBSM mon_right_id_rel.

Inductive mon_right_id_ext_rel : relation expr :=
| mon_right_id_ext_bound : ∀ (a : expr) (x : value_name) (t :
   typexpr),
  mon_right_id_ext_rel
  (E_live_expr (a) >>= (E_function x t (E_ret (E_ident x))))
  (E_ret a)
| mon_right_id_ext_orig : ∀ (a b : expr) (x : value_name) (t :
   typexpr), mon_right_id_rel a b → mon_right_id_ext_rel a b.

Theorem mon_right_id_ext_wbsm : WBSM mon_right_id_ext_rel.

Theorem mon_right_id_ext_rel_vewbsm : VEWBSM mon_right_id_ext_rel.
```

Listing B.5: Monadic left and right ret neutrality

```
Definition exprClosedOnVariable : expr → value_name → Prop :=
fun e v ⇒  ( ∀ (e' : expr), (subst_expr e' v e) = e ).

Inductive mon_assoc_rel : relation expr :=
| mon_assoc_start : ∀ (e e' e'' : expr) (v v' v'' : value_name)
(t t' : typexpr),
  v <> v'' → v' <> v'' → exprClosedOnVariable e' v'' →
  exprClosedOnVariable e'' v'' →
   mon_assoc_rel
   (e >>= (E_function v t e') >>= (E_function v' t' e''))
   (e >>= (E_function v'' t (E_apply (E_function v t e')
     (E_ident v'') >>= (E_function v' t' e''))))
| mon_assoc_s1 : ∀ (e e' e'' : expr) (v v' v'' : value_name) (t t' : typexpr), v <> v'' → v' <> v'' → is_value_of_expr e → exprClosedOnVariable e' v'' → exprClosedOnVariable e'' v'' →
    mon_assoc_rel ( (E_apply (E_function v t e') e ) >>= (E_function v' t' e'')) (E_apply (E_function v'' t (E_apply (E_function v t e') (E_ident v'') >>= (E_function v' t' e''))) e)
| mon_assoc_s2 : ∀ (e e' e'' : expr) (v v' v'' : value_name) (t t' : typexpr), v <> v'' → v' <> v'' → is_value_of_expr e → exprClosedOnVariable e' v'' → exprClosedOnVariable e'' v'' →
    mon_assoc_rel ( ( ( (subst_expr e v e'))  ) >>= (E_function v' t' e''))    ( ( ((E_apply (E_function v t e') e) >>= (E_function v' t' e''))) )
| mon_assoc_s3 : ∀ (e e' e'' : expr) (v v' : value_name) (t t' : typexpr), is_value_of_expr e →
    mon_assoc_rel ( ( ( (subst_expr e v e'))  ) >>= (E_function v' t' e''))    ( ( ((subst_expr e v e') >>= (E_function v' t' e''))) )
| mon_assoc_fin : ∀ (e : expr) ,
    mon_assoc_rel e e.
```

Listing B.6: Monadic associativity

B.4 Fork and join

```
Definition swapbodyl : expr :=
            E_ret (
             E_taggingright (
              E_pair (E_proj2  (E_ident (1)))
                     (E_proj1  (E_ident (1)))
                            )
                     ).

Definition swapbodyr : expr :=
            E_ret (
             E_taggingleft (
              E_pair (E_proj2  (E_ident (2)))
                     (E_proj1  (E_ident (2)))
                           )
                     ).

Definition swapbody : expr := E_case (E_ident (0))
           (1) (swapbodyl)
           (2) (swapbodyr).

Definition swapf : expr :=
    E_function (0) TE_unit swapbody.
```

Listing B.7: The original Coq *swapf* function

```
Inductive fork_comm_rel :  relation expr :=
 | forkee_start : ∀ (e e' : expr), fork_comm_rel (e # e') (e' # e)
 | forkee_endl : ∀ (e e' : expr), is_value_of_expr e → fork_comm_rel
    (e <# e') (e' #> e)
 | forkee_endr : ∀ (e e' : expr), is_value_of_expr e' →
   fork_comm_rel (e #> e') (e' <# e).

Theorem fork_comm_wbsm: WBSM fork_comm_rel.
```

Listing B.8: Fork commutativity

B.5 Deadlock properties

```
Definition takeright : expr :=
    E_function (0) TE_unit
      (E_case (E_ident (0))
           (1) (E_proj2  (E_ident (1)))
           (2) (E_ret (E_proj2  (E_ident (2))))).
```

Listing B.9: The original Coq *takeright* function

B.6 Equivalence

Appendix C

Project Proposal

C.1 Introduction of work to be undertaken

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

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

There have been two very successful frameworks, LWT[2] and Async[43] that both provided the primitives for concurrent development in OCaml however neither is supported by a clear semantic description as their main focus was ease of use and speed.

C.2 Description of starting point

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

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

C.3 Substance and structure

The project will consist of first creating a formal specification for a simple monad that has three main operations bind, return and choose. The behaviour of these operations will be specified in a current semantics tool like Lem[1] or Ott[40].

As large amount of research has gone into both monadic concurrency and implementations in OCaml, the project will draw inspiration from Claessen[15], Deleuze[17] and Vouillon[45].

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

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

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

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

C.4 Criteria

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

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

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

C.5 Timetable

The project will be split into two week packages

C.5.1 Week 1 and 2

Preparatory reading and research into tools that can be used for writing the specification and in the extension, the proofs. The tools of choice at the time of proposal are Ott for the specification step and Coq[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.

C.5.2 Week 3 and 4

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

Deliverable: A document describing the major design decisions of the two libraries, the difference in design of the specification and a set of test cases much like the ones used in OCaml Light [34, 4], but with a concurrency focus.

C.5.3 Week 5 and 6

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

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

C.5.4 Week 7 and 8

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

C.5.5 Week 9 and 10

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

Deliverable: Evaluation data and charts, the halfway report.

C.5.6 Week 11 and 12

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

C.5.7 Week 13 and 14

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

C.5.8 Week 15 and 16

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

C.5.9 Week 17 and 18

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

C.5.10 Week 19 and 20

Finalising the dissertation and incorporating all feedback and polishing.

C.6 Resource Declaration

The project will need the following resources:

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

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

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

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

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