Tamás Kispéter

Monadic Concurrency in OCaml

Part II in Computer Science

Churchill College

April 30, 2014

Proforma

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

Original Aims of the Project

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

Work Completed

All that has been completed appears in this dissertation.

Special Difficulties

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

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

Declaration

I, Tamás Kispéter of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date April 30, 2014

Acknowledgements

Chapter 1

Introduction

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

1.1 Motivation

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

Motivation of the project is to investigate the lack of certified implementation of a concurrency framework. Verified concurrent systems have been researched for languages like Csevvcik2011relaxed, C++ and Javalochbihler2012machine, 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. LWTLWT and AsyncAsync. They both provide the primitives and syntax extensions for concurrent development. Neither is supported by a clear semantic description, because their main focus is ease of use and speed .

LWT, the lightweight cooperative threading libraryvouillon2008lwt 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.

In **??** we can see some of the syntax of LWT. Lines **??**–**??** define heads, a function that sleeps for 1 second and then prints "Head", and lines **??**–**??** define tails which sleeps for 2 seconds and then prints "Tails". Lines **??**–**??** 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.

In **??** we define `heads` and `tails` as Deferred values of the respective code sequences. A Deferred is an implementation of a promise.

There are a number of other experimental implementations of concurrency in OCaml. For example JoCamljocaml implements join calculus over OCaml, Functoryfunctory focuses on distributed computation, OCamlNet exploits multiple cores and OCamlMPIocamlmpi 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 concurrencydeleuzelight are capturedfriedman1988applications and delimitedkiselyov2010delimited continuationsshan2004shift, trampolined styleganz1999trampolined, continuation monadsClaessen99functionalpearls, promise monadsliskov1988promises and event based programming (as used, for example in the OCamlNetOcamlnet project). This work focuses on the continuation monad style.

A monadhoareetal2001tackling in functional programming is a construct to structure computations that are in some sense "sequenced" together. This sequencing can be for example string concatenation, simple operation sequencing (the well known semicolon of imperative programming) or conditional execution. Two operations commonly called bind and return and a type constructor of a parametric type, like $\alpha\, M$ where α is any type, form a monad when they obey a set of axioms called monadic laws.

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

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

1.5 Semantics to logic

The semantics of concurrency can be modelled in logic, in particular logics used by proof assistants. The developer can use this model to formally verify properties about the semanticsbenton2008mechanized,blazy2009mechanized,blazy2006formal,leroy2009formal. CoqCoq, HOL and Isabelle are widely used proof assistants. Tools like OttOtt help with the modelling process with ascii-art notation and translation to proof assistants and LaTeX.

1.6 Logic to runnable code

While a number of proof assistants have utilities for direct computation, in most cases semantics is described as a set of logical, not necessarily constructive relations. This representation is more amenable to proofs than to actual execution, because there is no need for an input-output relationship. Without this strict requirement on the relation the representation can be more succinct, but hard to extract. Letouzeyletouzey2008extraction has shown that many such definitions can be extracted into executable OCaml or Haskell code. Coq and Isabelle provide tools for this extraction. The tools also generate a proof that the extracted code is faithful to the representation in the proof assistant.

Chapter 2

Preparation

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

2.1 Design of concurrent semantics

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

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

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

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

2.2 Choice of implementation style

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

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

2.3 Design of monadic semantics

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

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

1. $$\mu \circ T\mu = \mu \circ \mu T$$

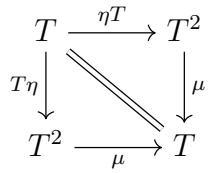
   Or as commutative diagram:

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

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

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

Or as commutative diagram:



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

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

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

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

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

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

1. ret is essentially a left neutral element:
$$(\mathbf{ret}\,x) \gg= f \quad \equiv \quad f\,x$$

2. ret is essentially a right neutral element:
$$m \gg= \mathbf{ret} \quad \equiv \quad m$$

3. bind is associative:
$$(m \gg= f) \gg= g \quad \equiv \quad m \gg= (\lambda x.(f\,x \gg= g))$$

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

2.4 Tools

The project uses a chain of three tools:

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

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

2.4.1 Ott

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

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

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

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

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

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

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

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

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

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

Finally paramount to most semantics are relations like the reduction relation. In **??** I define a set of mutually recursive relations named Jop with one relation in it the $--> $ or reduce relation. Each element of this relation takes the form t1 $--> $ t2, where t1 and t2 are both terms of the grammar defined above. There are three statements for function application: the actual substitution, reduction of the first term and reduction of the second term. The premise(s) appear line-by-line above the ascii-art line, while and the result below the line. Next to the line is the name of the statement which is then prefixed by the name of the relation to avoid ambiguity.

2.4.2 Coq

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

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

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

Predicates like equality and other sets can be used as well.

New predicates can be defined inductively

Data structures can also be defined both inductively and coinductively.

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

Finally theorems can be proven with these propositions and structures.

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

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

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

However with the help of a plugin developed by David Delahaye, Catherine Dubois, Jean-Frédéric Étienne and Pierre-Nicolas Tollitte delahaye2007extracting,tollitte2012producing by marking different modalities of the inductively generated proposition we can generate code with an input-output convention.

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

2.4.3 OCaml

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

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

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

Chapter 3

Implementation

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

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

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

3.1 The semantics

In this section I will describe the implemented semantics. These semantics were written in Ott. Much of the inspiration for the semantics of basic features comes from Benjamin C. Pierce: *Types and Programming Languages*pierce2002types.

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

The language of this concurrency framework is simply typed.

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

The semantics splits into three main parts

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

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

I will detail the semantics feature-by-feature: arrow types or functions, sum types or tagged unions, product types or pairs, the fixpoint combinator, the monadic primitives and the fork operator. The presentation of semantics for each of the features were inspired by Piercepierce2002types. Each section will have a corresponding table of syntax, evaluation rules and typing rules. In the syntax section I will introduce every new syntactic form used in the evaluation and typing rules, but I will not repeat previously mentioned syntax. The presentation of both the syntax and the evaluation and typing rules were slightly simplified for better readability.

3.1.1 Arrow types

Arrow types or functions are ubiquitous in functional programming languages. In **??** I detail the basic syntax and semantics of functions. The style and details are based on Pierce[p. 103]pierce2002types. A function abstraction $\lambda x : T.e$ encloses a yet unreduced expression e that may involve the variable x that is bound by the abstraction. I used call-by-value semantics for functions and head-first reduction. Furthermore, arguments are explicitly annotated in functions and substitution does not provide facilities for renaming. It is presumed that the user of the framework takes care of picking fresh variable names.

3.1.2 Sum types

Many circumstances require the ability to describe expressions that are either one type or the other. Sum types or otherwise known as labelled unions are a simple solution to this. An expression e that has type T can be labelled as a variant in a sum type by attaching a label **left** e and **right** e will give force it to have type $T + T'$ and $T' + T$ for some type T'. This expression can be then destructed by the expression "Case e of **left** $x_1 \Rightarrow e_2$ | **right** $x_2 \Rightarrow e_3$ " which will evaluate to either a substitution of e_2 or e_3 depending on what the tag said. In this project I use sum types in the signature of **fork** that will be detailed later. In **??** I detail the implementation of sum types that is based on Pierce[p. 132]pierce2002types.

3.1.3 Product types

One very common feature of functional languages is pairs or product types. Grouping different types of data together in one logical unit often makes code more simple. For example computations with complex numbers would be rather unintuitive if we always had to handle the real and imaginary parts separately. To define products, first I introduce the syntax $\{e, e'\}$, the pair of expressions e and e'. If they had types T and T' respectively I define the type construct for this $T \star T'$. I have introduce three primitive functions to deal with pairs: **pair** places two values in a pair, **proj1** takes the first element of a pair and **proj2** takes the second. In **??** I describe the precise implementation of product types defined in this language. The style and details are based on Pierce[p. 126]pierce2002types.

3.1.4 Fixpoint combinator

Recursion is very characteristic of functional programming languages. There are many ways to achieve recursive constructs in terms and even in types. A very

elegant treatment surfaced from the formal study of recursive constructs in the form of fixpoint combinators. In denotational semantics for example loops and other recursions are treated as the greatest fixpoints of continuous functions. A fixpoint combinator y is defined as a term that given function f satisfies

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

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

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

However in languages like the one implemented in this chapter, where arguments are always reduced before the function application can begin the Y combinator would always diverge. There are more than one option for implementation in languages like this, but I have chosen to use the one described in Pierce[p. 144]pierce2002types. This style fits well with the tool chain I used as it does not require another partial function application and it is also very minimal.

3.1.5 Monadic primitives

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

- ret, also known as return. It has type $T \to$ **con** T and simply evaluates its parameter and boxes up the result in the parametric type
- >>=, also known as bind, sequences two operations. The second argument is the continuation for the first parameter. More formally it has a type **con** $T_1 \to (T_1 \to$ **con** $T_2) \to$ **con** T_2, that is it takes a boxed up computation and a function that takes the value of the computation and returns a new box. Bind then evaluates the expression within the box of the first argument and passes it to the second argument.

3.1.6 Fork

Up until this point I have not mentioned any language feature that implements concurrency which is the main focus of the dissertation. The third, additional

operation of the concurrency monad is fork, which is the way to spawn new threads (two in this particular case). There are a number of ways to implement fork and indeed this project went through various iterations of semantics for this operation.

As a first approximation I had to decide on a signature for fork. On the argument side fork may take zero, one, two or many arguments. Zero 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, Joneshoareetal2001tackling chose an implementation with one argument, however with a similar requirement of a metalanguage of parallel terms $e \mid e'$. I chose to go with two arguments as that can maintain a simple evaluation context with reductions from language expression to expression. The choice between these three argument styles is largely arbitrary as they all implicitly form the binary parallel composition $e \mid e'$. Note however that several arguments can be elegantly simulated by composing binary parallel compositions.

To stay within the monad I have chosen to have the arguments as already boxed terms, that is of type $\mathbf{con}\ T_1$ and $\mathbf{con}\ T_2$ for some T_1, T_2. For better expressibility, I chose to take the two argument curried that is

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

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

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

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

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

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

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

3.1.7 Computation place holders

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

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

3.2 Proof assisstant system

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

3.2.1 Outline of the proof assistant code

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

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

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

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

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

4. Lemmas: There are a few supporting lemmas about the equality of variables. These are automatically generated by Ott. Lemmas are denoted

with diamonds. An easy example would be the straightforward lemma that says labels are either equal or not in **??**.

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

3.2.2 Extractable reduction relation

Coq provides extraction facilities to OCaml and Haskell. However, the built in extraction only deals with inductive sets and fixpoints that do not involve propositions, more specifically the *Prop* sort. There are good reasons why logical inductive definitions are not extracted from Coq. Logical inductive types do not need to conform to any input/output relationship and computations that correspond to a logical inductive relation need not always terminate. Extraction from Coq is required to produce a certification of equivalence and the Calculus of Inductive Constructions, the logic underlying Coq, cannot directly express a certification for a non-terminating program.

Logical inductive types are often used for description of various concepts in semantics, especially reduction relations. Ott generates a logical inductive relation as seen in **??**. For simple relations, like the value relation in **??** it is simple to rewrite in a set inductive manner. However, for complex non-terminating relations, like the reduction relation in this project, rewriting is not feasible. Delahaye et al.delahaye2007extracting,tollitte2012producing proposed a method of extracting such relations to OCaml by annotating the input/output modes of the elements of the relation.

I rewrote the value check logical fixpoint to an extractable version by simply replacing the True and False propositions by their boolean counterpart and successfully extracted it by the built-in Coq extraction. However, not even the Coq plugin based on element modes could extract the reduction relation generated by Ott.

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

program.

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

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

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

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

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

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

3.3 OCaml system

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

3.3.1 Outline of the OCaml code

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

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

Their interdependence is shown on **??**.

3.3.2 Hand modifications and justifications

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

Furthermore I have changed the way `selectstar` is implemented. The extraction results in the type in **??** 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 **??**.

As the extraction plugin is experimental there were a number of inefficiencies and a few issues due to the fact that the semantics of OCaml pattern matching is sequential, while it is parallel in Coq. While Tollittetollitte2012producing made progress on the merging of cases when one case subsumes the other, these were not always correctly identified. For example the case **??**–**??** in **??** subsumes the case **??**. The plugin was unable to infer this as that would have required it to show that function term is always a value, regardless of its parameters. This is

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

My solution was to insert the correct step into the failing match as in **??**. Note, that taking that reduction may fail, but that is the expected behaviour. A reoccurring inefficiency comes from the extraction plugin generating a default failing case in all situations, even if the default case may never happen. **??** in **??** is an example of this: a boolean may only take the values true or false. A similar issue occurs when evaluating a function: **??** in the same listing matches by giving a name to the return value, but generates a default case as well. To this latter problem I simply removed the match and replaced the name of the return value with the function call.

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

Chapter 4

Evaluation

4.1 Theoretical evaluation

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

4.1.1 Methods

Weak bisimilarity

Intro to weak bisimilarity.

What form does a general weak bisimilarity proof take.

How does it appear here.

4.1.2 Properties

Monadic laws

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

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

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

Fork commutativity

Outline of fork commutativity

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

Fork associativity

Fork in this implementation is not associative:

Deadlock properties

$$\delta \,|\, x \quad \simeq \quad x$$

$$\delta \gg\!= e \quad \simeq \quad \delta$$

Remarks on congruence

Type preservation?

Progress?

4.2 Practical evaluation

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

4.2.1 Methods

4.2.2 Examples

Kahn process network

Eratosthene Sieve

Concurrent sort

Chapter 5

Conclusion

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