Formally verified programming with monads in Coq

(Formalnie zweryfikowane programowanie z monadami w Coqu)

Zeimer

Praca inżynierska

Promotor: prof dr rehabilitowany Wpisuyashi TODO

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

wrzesień 2019

Abstract

We introduce *hsCoq*, a Coq library for formally verified programming with Haskell-style abstractions: functors, applicative functors, monads, monad transformers and typeclass-based effects. We discuss the design choices we made and illustrate the working of the library by formalizing examples taken from [1].

Prezentujemy *hsCoq*, Coqową bibliotekę do formalnie zweryfikowanego programowania z abstrakcjami w stylu Haskella: funktorami, funktorami aplikatywnymi, monadami, transformatorami monad oraz efektami opartymi o klasy typów. Omawiamy nasze decyzje projektowe i przedstawiamy działanie biblioteki na przykładach z [1].

Contents

Chapter 1

Introduction

This chapter gives some motivations for why formal verification of hardware, software and mathematics is useful and then briefly introduces the Coq proof assistant – a tool for such verification – to those who are not familiar with it.

The rest of the thesis is structured as follows:

- In chapter 2 we discuss the problem of modeling computational effects in programming languages and compare existing approaches.
- In chapter 3 we present our library *hsCoq* and discuss its design.
- In chapter 4 we give some example effectful programs and prove their properties.
- In chapter 5 we describe our approach to proof engineering (the formalized mathematic's equivalent of software engineering) in our library.
- In chapter 6 we conclude and give some possible directions for further work.

1.1 Formal verification of hardware and software

Since their invention in the 1940s, computers' significane rose at a very fast pace. They were getting applied to an ever expanding range of problems by more and more people, private companies and governments alike. It shouldn't be a considered a surprise then that we became very reliant on them for both small conveniences and large scale projects.

But significane is not the only thing that rose – another one is complexity. Exponentially growing processing speed required the complexity of chip designs to grow at a similar rate. More complex products and services require more complex software architectures and with new business models, like cloud computing, comes even more complexity in the form of virtualization, containerization and so on.

And with complexity comes, of course, the potential for bugs, which may cause a lot of damage. A malfunction in software running the stock exchange can mean billions of dollars of losses (if not an accidental global recession!); in software running a nuclear power plant – deadly radiation for thousands of people and energy shortage for millions more.

Due to these dangers a lot of effort has been put into assuring that hardware and software are correct and with great success, but here and there bugs still have crept in. Some of the most spectecular were, recently:

- Metldown, which "exploits side effects of out-of-order execution on modern processors to read arbitrary kernel-memory locations including personal data and passwords." [2]
- Spectre, which uses speculative execution and branch prediction to "leak the victim's confidential information via a side channel to the adversary." [3]
- Heartbleed, which "allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet" [4]

Academia and industry, however, are not passively waiting for more disasters to happen. DeepSpec [5] is a Coq-based project that tries to eliminate both hardware and software bugs and security vulnerabilities by creating a web of formally verified hardware, operating systems, compilers, web servers, cryptography libraries etc. Even though mildly successful already, a long time will pass until it will finally help verify mission-critical real-life systems, like the software running Boeing 737 MAX, which recently caused two such planes to crash [6].

1.2 Formal verification of mathematics

Hardware and software are not the only things in need of formal verification – mathematics is also one of them.

The four colour theorem is a problem posed in 1852. It states that any planar map can be coloured with only four colours so that no two regions sharing a boundary are assigned the same colour. It became famous for resisting many proof attempts by many famous mathematicians for more than a century until it was finally proved by Appel and Haken in 1976. Its importance stems from the proof method – it was the first major theorem proven using a computer program, whose job was to make sure a very large case analysis was exhaustive.

Thomas Tymoczko, a philosopher of mathematics, criticised this proof by labeling it with a term he invented just for this purpose – "non-surveyable". He considered a proof to be non-surveyable when its verification cannot be performed

by human mathematicians competent in the relevant field. Appel and Haken's proof certainly did fail the surveyability criterion – the program was written in IBM 370 assembly, a language graph theorists very likely didn't understand.

This was the perfect theorem to let formal proofs and formally verified programs shine by dispeling Tymoczko and other mathematician's doubts. This is what indeed happened in 2005, when Georges Gonthier presented a proof of the theorem formalized in Coq [7] [8].

But big theorems with difficult proofs are not the only call for formalized mathematics. Another one is the mere fallibility of humans, especially their limited memory and reasoning skills and the tendency to follow authority. As unmathematical as it sounds, these are the main reasons cited by Vladimir Voevodsky, a field medalist mathematician turned a fan of formal proofs, in one of his talks [9].

The fields he refers to are homotopy theory, higher category theory and motivic cohomology – all of them containing many layers of abstraction, tons of concepts and definitions, and based on rather shaky foundations (in the sense of foundations of mathematics).

As an example, noticing an error in one of his papers took 7 years and repairing the mistake another 6 years. In another, more extreme case, after publishing a paper in 1989, an alleged counterexample was found in 1998 by another expert in the field, but it was too difficult for them to agree on whether it really was a counterexample and Voevodsky only realized he was wrong in 2013. All of this put him in search of formalized foundations of mathematics, and he chose Coq to purse them.

1.3 The Coq proof assistant

Coq [10] is a proof assistant that was started in the late 1980's in France and is still under active development. It consists of three complementary languages:

- Gallina, the term language, implements a formal system whose slight variants go under a plethora of names: Calculus of (Inductive) Constructions, (Intensional) Martin-Löf Type Theory, Intuitionistic Type Theory, Constructive Type Theory, etc. We can use it to express specifications, programs, theorems and proofs.

- Vernacular, the command language, is a language of commands, which allow things like looking up useful theorems in the environment or creating modules.

- Ltac, the tactic language, is a language which facilitates writing proofs – these can in principle be written using the term language, but it's unwieldy even for simple proofs.

The basic objects of our interest in Coq (and in any kind of the aforementioned formal systems) are types. Their role is to classify terms – for example, `21 + 21` is a term of type `nat`, written `21 + 21 : nat`. Types are syntactical entities, which means that the judgement `x : A` can always be checked algorithmically.

Thanks to the the Curry-Howard correspondence [11], types can seen both as specifications of programs and as statements of theorems. All programming and proving may be then conceptualized as manipulating a few kinds of rules:

- Formation rules tell us what types are there.
- Introduction rules tell us how to construct elements (canonical terms) of a given type.
- Elimination rules tell us how to use an element of some type to construct elements of other types.
- Computation rules tell us how an elimination rule acts on an introduction rule – computation happens when we first build something and then take it apart.
- Uniqueness rules tell us how an introduction rule acts on an elimination rule – if we first take something apart and then rebuild it, we should get the same thing we initially had.

How does this play out in practice? Let's take a look at an example development, which illustrates the basic workings of Coq. At this point, we strongly encourage the unfamiliar reader to install CoqIDE (a dedicated IDE for Coq, available from [10]) and run this snippet (which can be found in the thesis sources in the directory `snippets/`) in interactive mode – this experience will be better than any explanation.

```
1  Print nat.
2  (*
3  Inductive nat : Set :=
4      | O : nat
5      | S : nat -> nat.
6  *)
7
8  Inductive Tree (A : Type) : Type :=
9      | Leaf : A -> Tree A
10     | Node : Tree A -> Tree A -> Tree A.
11
12 Arguments Leaf {A} _.
13 Arguments Node {A} _ _.
14
```

```
15 Fixpoint label
16   {A : Type} (t : Tree A) (n : nat) : nat * Tree (A * nat) :=
17 match t with
18     | Leaf x => (n, Leaf (x, n))
19     | Node l r =>
20         let (n', l') := label l n in
21         let (n'', r') := label r (S n') in
22           (n'', Node  l' r')
23 end.
24
25 Definition lbl {A : Type} (t : Tree A) : Tree (A * nat) :=
26   snd (label t 0).
27
28 Compute lbl (Node (Node (Leaf true) (Leaf true)) (Leaf false)).
29 (* = Node (Node (Leaf (true, 0)) (Leaf (true, 1))) (Leaf (false, 2))
30    : Tree (bool * nat) *)
31
32 Fixpoint size {A : Type} (t : Tree A) : nat :=
33 match t with
34     | Leaf _ => 1
35     | Node l r => size l + size r
36 end.
37
38 Require Import Arith.
39
40 Theorem label_size :
41   forall (A : Type) (t : Tree A) (n n' : nat) (t' : Tree (A * nat)),
42     label t n = (n', t') -> S n' = n + size t.
43 Proof.
44   induction t as [| l IHl r IHr]; intros.
45     rewrite <- plus_comm. cbn. inversion H. reflexivity.
46     cbn. intros.
47       case_eq (label l n); intros m1 t1 H1.
48       case_eq (label r (S m1)); intros m2 t2 H2.
49       rewrite H1, H2 in H. inversion H; subst.
50       rewrite (IHr _ _ _ H2), (IHl _ _ _ H1), plus_assoc. reflexivity.
51 Qed.
```

A Coq file consists of a series of commands, which define types and terms, import and export modules, state or look up theorems etc. In this file we want to implement a function that labels the leaves of a tree with natural numbers starting

from 0.

For this, we will need the type `nat`, which is provided by the standard library. We can print its definition using the command `Print`. What we get is an inductive definition of a type with two constructors, `O` and `S`. `O` is supposed to represent 0 and `S` is supposed to represent the successor operation, so that `S O` represents 1, `S (S O)` represents 2 and so on.

We will also need a type `Tree` that represents trees. The definition will be inductive, just as for natural numbers, but this time is has a parameter A : `Type` that tells us the type of elements that can be stored in the tree. This can be seen as defining infinitely many types at once – one for each choice of A. There are two constructors: `Leaf`, which represents a tree with one element, and `Node`, which represents a tree with two subtrees. Thus our trees are always nonempty and they contain elements only in their leaves.

The next two commands, `Arguments`, control the use of implicit arguments. Normally, we would have to write `Leaf nat 42` for the tree containing only the value 42, but this is redundant, since given a term like `42`, we can infer its type to be `nat`. We can thus write this as `Leaf _ 42`, but this is still redundant. Thanks to the first of these two commands, we can write simply `Leaf 42`. The second command has a similar effect on the other constructor, `Node`.

The function `label` is the clou of the whole development. It's type says that given a type `A` (which we won't need to give explicitly, since it's an implicit argument, thanks to the curly braces), a tree `t` that holds elements of type `A` and a natural number `n` which acts as the counter, it returns a pair consisting of a natural number (the new counter) and a tree containing elements of type `A * nat`.

The definition is recursive (marked by the keyword `Fixpoint`). When we hit a `Leaf`, we return the current counter and label the leaf with the counter's value. When we encounter a `Node`, we label the left subtree using the current counter, then label the right subtree with the new counter incremented by one, and then return the new counter and the two labeled subtrees joined by the constructor `Node`. We can use this auxiliary function to define `lbl`, the function we wanted from the very beginning.

The command `Compute` normalizes a given term, which in type-theoretical parlance means just running a computation (which in Coq is guaranteed to finish). In our case, we run `lbl` on an example tree to see that the result is as we expected – the leaves are now labeled with natural numbers, starting at zero, from left to right.

But how do we prove what we have done really is what we intended? The basic approach is to invent some properties we would like our program to satisfy. If it does satisfy all of these desired properties and they encapsulate everything we require from the program's behaviour, we may consider the program correct for our purposes.

In our case, one simple property is that after labeling a tree `t` the counter increases by the size of the tree minus one. To state this property formally, we first need to define a function that computes the size of a tree. We do this by recursion: a `Leaf` has size 1 and the size of a `Node` is the sum of sizes of its subtrees.

The command `Require Import` imports a module. To prove our theorem, we will need some simple theorems about addition and multiplication of natural numbers, which we can find in a module called `Arith`.

Now we are ready to state our theorem. It reads: for any type `A`, any `t` which is a tree of elements of type `A`, any two natural numbers `n` and `n'` and any `t'` which is a tree of elements of type `A * nat`, if calling label on `t` and `n` results in the pair `(n', t')`, then the successor of `n'` is equal to `n + size t`.

There are some points to be noticed. First, we quantify over types, which means that our logic is not first-order, but higher-order. Second, we need to quantify over `t'`, even though the tree resulting from the call to label is of no interest to us. Third, we avoid using subtraction or the predecessor function and instead express our theorem using successor. This is a technicality – our theorem will be a bit easier to prove this way.

We now enter the proof mode, in which we can use tactics to prove our theorem. The command `Proof` is useless, because it does nothing, but it looks nice at the beginning of a proof.

We proceed by induction on `t`. The clause `as [| l IHl r IHr]` allows us to name variables and induction hypotheses. This may seem weird to a classically-minded mathematician, but is actually an important aspect of proof engineering – to make our proofs readable and resilient to change, it's a good practice to name variables and hypotheses manually (as opposed to using automatically generated names).

To boost the unfamiliar reader's curiosity and encourage him to see the proof in CoqIDE, we will not go over it in detail. It suffices to say that the in first case the result holds almost by computation, but we need to use the commutativity of addition and in the second case we basically just use our two inductive hypotheses and the associativity of addition.

This concludes our introduction to the Coq proof assistant. From now on we will assume that the reader is familiar with it. If that's not the case, there are a great many of books which make a good introduction:

- Software Foundations [12] is a four volume book by many authors. The first volume treats the basics of Coq and the third one is about formally proving correctness of algorithms in Coq. The second and fourth volumes are less relevant, as they are about formalizing programming languages and randomized testing, respectively.

- Coq'Art [13] is a comprehensive but a little dated (2004) book that covers way more material than Software Foundations, including coinduction, proof by reflection, case studies and so on.
- Certified Programming with Dependent Types [14] is an advanced book that focuses on using dependent types and proof engineering, but also covers topics like axioms, reasoning about equality proofs and general recursion.

Chapter 2

Effects

2.1 Basic concepts

Although *hsCoq* is a rather general-purpose library, it also tackles the problem of how to best express (and program with) effects. The research on effects is still young and active, so in this chapter we will take a look at the basic concepts to gain some familiarity with it. However, to keep the exposition accessible to ordinary programmers, our treatment will be rather loose and hands-on.

A **side effect** is some kind of communication with the outside world, like:

- Reading external configuration.
- Logging to a file.
- Pseudorandomness, because it requires a seed.
- Any kind of IO, like connecting to the Internet or querying a database.
- Memory management, like allocation and freeing.
- Concurrency and threads, because they need some kind of synchronization, either through shared memory or message passing.

An **effect** (sometimes also called a **computational effect**) is either a side effect or use of some control mechanism, like:

- Generators and coroutines.
- Nondeterminism: the computation may return multiple results.
- Partiality: the computation may return `null` or some similar value representing the lack of result.

- Exceptions: the computation may fail by throwing an exception, which may or may not then be handled by some other computation.
- Goto: the computation may pass control to some other computation.
- Continuations: the most general control mechanism, which can express all of the above ones.
- Sometimes nontermination and even termination are also considered effects, but that's a tougher topic, beyond the scope of this thesis.

It should be quite obvious that effects are useful. They are in fact the main reason for running most computer programs and without them running even the simplest number-crunching program would be pointless, since without IO operations it couldn't tell us the result. We thus want to be able to express effects. But we also want to be able to reason about our effectful programs, which in most programming languages is notoriously difficult, and moreover we want to verify our resaoning.

2.2 A comparison

To better understand the matter, we can classify all expressions in a programming language as being either values or computations. A **value** necessarily has no effects whereas a **computation** may have effects. Philosophically, we say that a value is but a computation does. We can then say that a language is **pure** when it explicitly separates values from computations and **impure** when it does not.

When interpreted as a yes-no property, most languages are impure, so it is better to see the concept of purity as a continuum, where some languages may be more pure than others. Let's examine a few real-world languages and see, where they can be placed on this continuum.

C is a very impure language. There is global state in the form of global variables. Even though "considered harmful", we can use the `goto` statement anywhere, just as IO operations. It's fair to say that C certainly does not even try to separate values from computations.

In Java, there's no global state, but objects still have internal state that methods can change. There's no `goto` and some exceptions are checked, which makes them apparent in type signatures. But there also are unchecked exceptions that don't appear in type signatures and IO operations may be used anywhere. Thus, even though there are some ad hoc attempts at separating values from computations (checked exceptions), Java still can't do this in a principled way. It is still an impure language, though less than C.

Haskell, on the other hand, is a reasonably pure language. There is no global state. There's an impure error mechanism, but it is barely used and usually re-

placed with explicit error handling. IO can be used anywhere by using functions like `unsafePerformIO`, but this can't be done accidentally and is generally avoided. Therefore, nearly all error handling and IO and all other effects are expressed using monads, which are clearly visible in type signatures and make values and computations very easy to tell apart. This is the reason that Haskell is generally perceived as a pure language.

At last, Koka [20] is one of the purest languages in existence. It clearly separates values from computations – each expression, besides its type, is also assigned the list of all effects that it may produce and even nontermination is tracked this way. The tracking is performed by the effect system and is an integral part of Koka's design. This may be contrasted with Haskell, where monads are a library-level concept and the languge was not designed with the value-computation distinction in mind.

To better see these differences, let's look at a piece of code that loops before trying to print something to the screen in each of these four languages.

```c
void f()
{
    f();
    printf("Effectful\n");
}
```

Listing 1: C

```java
public void f()
{
    f();
    System.out.println("Effectful");
}
```

Listing 2: Java

```haskell
f :: IO ()
f = do
    f
    putStrLn "Effectful"
```

Listing 3: Haskell

```
function f() : <div, io> ()
{
    f()
    println("Effectful")
}
```

Listing 4: Koka

We see that in C and Java the return type is just `void`, which in these languages represents a type with only one value. There is no type-level indication whatsoever that these programs perform any kind of side effects.

In Haskell, the type is `IO ()`, where `()` is analogous to `void` and `IO` indicates that an IO operation. The `IO` part can't be omitted in the type of `f`, because the type of `putStrLn` is `String -> IO ()`.

In Koka, the return type is also `()`, which is analogous to Haskell's `()` and C and Java's `void`, but the signature also contains the row of effects `<div, io>`, which indicates that `f` can loop or perform IO operations.

We can see that, even though at their foundations those four languages are quite different from each other, C and Java cluster together at the impure end of the spectrum, whereas Haskell and Koka, despite their effect systems being different, cluster together at the pure end.

2.3 Approaches to effect systems

An **effect system** is a means of carrying out the separation of values and computations. In the previous section we have already seen two approaches to effect systems, but before taking a closer look at them, we should first ask whether it is beneficial to have an effect system. This question is best answered by Tony Hoare, who invented quicksort, Hoare logic, Communicating Sequential Processes and also the worst nightmare of many programmers, the null reference [15]:

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years

This "billion dollar mistake" is only the tip of the iceberg of allowing effects without properly tracking them (references allowing nulls can be seen as a kind of "missing value" effect). It isn't too hard to imagine that other effects can cause even more problems. For example, if a function throws an exception without this being indicated in the type signature, the client code may forget to handle it.

The problem of null references is easily fixed with some kind of effect system. For example, we could require all references to be non-null and introduce a special option type to represent missing values. Then references which can be "null" can be statically differentiated from those that cannot by the type system.

After seeing some anecdotal evidence supporting the superiority of having an effect system, we are ready to see the contending approaches.

2.3.1 Monads and their friends

The oldest approach, which is quite ad hoc, amounts to implementing monads and various related structures at the library level and using them to track effects using the type system. This approach can be called the `mtl` approach, after a Haskell library that implements it.

Monads were initially invented by category theorists in the 1960s for the purpose of universal algebra and named after a concept coming from the philosophy of Leibniz. A running joke in the functional programming community, originally due to Mac Lane [16], is that

> a monad in X is just a monoid in the category of endofunctors of X, with product $\times$ replaced by composition of endofunctors and unit set by the identity endofunctor. What's the problem?

Despite their obscure provenance and a reputation of being hard to understand [17], Moggi realized in his famous 1991 paper [18] that they are in fact quite a simple abstraction that can be used to represent effects. Namely, a monad $M : \texttt{Type} \to \texttt{Type}$ is a function from types to types, and $M\ A$ may be interpreted as the type of computations that return a value of type A, but which may have an effect represented by M.

Each monad has an operation $\texttt{pure} : A \to M\ A$ which turns a value into an effectless computation and an operation $\texttt{bind} : M\ A \to (A \to M\ B) \to M\ B$, which runs the first computation and feeds its result to a function that uses it to produce another computation. These operations are related by laws that are monoid laws in disguise.

Monads, however, are not the silver bullet of effectful programming and come bundled not only with operations, but also with problems. The most obvious problem is that a monad represents a single effect, but we of course want to have multiple

effects at once. Implementing a new monad for each desirable combination of effects is not a sane option and this causes us to search for a way of composing monads.

Such a way of composing monads are monad transformers [19]. A monad transformer `T` is an object of type `T` : (`Type` → `Type`) → (`Type` → `Type`) that represents an effect and `T M A` (`T` applied to a monad `M` and a type `A`), represents a type of computations that return a value of type `A` but may also perform effects represented by `T` and `M`. It comes bundled with an operation `lift` : `M A` → `T M A` that turns a computation that can have only one kind of effect into one that can have two.

Transformers `T` and `S` and a monad `M` can then be composed to give a monad `T` (`S M`). Because each monad has its corresponding transformer, we can get rid of all the monads (except for one, the identity monad) and use just the transformers.

Monad transformers, even though they solve the problem of composing monads, come with their own problems too. The simplest and most annoying is the fact that for complicated transformer stacks we need to use the `lift` operation a lot. For example, we would need two `lift`s to get from `M A` to `T` (`S M`) `A`. This is quite a big problem, because real-world code often requires more than just two effects.

This deficiency can be remedied by introducing a more abstract design based on typeclasses [19]. Here each effect `E` is split into a specification and an implementation. The specification is a typeclass expressing that a monad `M` supports an effect `E` and the various monad transformers can then implement this typeclass.

This solves the problem with too many `lift`s, but still has its own problems. One is that the number of implementations required is quadratic, because usually each transformer needs to implement each typeclass. However, in practice this is not as big a problem as having to implement a separate monad for each combination of effects.

Another problem, far bigger and more conceptual, is the ordering of effects: if a monad is both stateful and nondeterministic, how do these two effects interact? Is there a global state or does each branch of the computation have its own local state?

2.3.2 Algebraic effects and handlers

These problems with effect ordering suggest the idea that maybe there shouldn't be a static ordering. Maybe the meaning of combinations of effects, like state + nondeterminism, shouldn't be fixed ahead of time? Ultimately, maybe even the meaning of the effects themselves shouldn't be fixed?

This leads to a refinement of the notion of effect specification, which ceases to be an interface describing the operations associated with the effect and becomes an algebraic data type that represents these operations. Thus these new effects can be

called **algebraic effects**, but there is more to this name than just the algebraicness of the data types.

This change prompts a refinement of the notion of implementation of an effect. Instead of simply being a monad transformer, an implementation of an effect is now an **effect handler** – an operation that takes an effectful computation as input and transforms it into another computation. We can use handlers to realize an effect in some concrete way, translate the effect into some other effect or even throw the effect away.

For example, when we have a "log" effect and a "debug" effect, our production code can realize logging using some logging service and discard the debugging effect while our test code can do exactly the opposite! This can be useful for testing in an even more general way: the test code can interpret effects by mocking them, which is a very popular unit testing technique in the object-oriented programming world.

This approach has two flavours. The first one aims to implement these algebraic effects and handlers at the library level using concepts like free monads or open unions. [**?**] [**?**]

WUUUUUT jedziem do lasu na grzyby

The most recent and promising approach, represented by languages such as Koka [20], Eff [21], Frank [22], Multicore OCaml [23] and Helium [24], is called algebraic effects

2.4 Effects in Coq?

At long last, Coq is a completely pure language. This is not because of some ingenuity on the party of its creators – it's just that Coq has no IO and, more generally, no way to communicate with the outside world. This shouldn't surprise us – as a proof assistant it's main purpose is to prove theorems, not to write programs interacting with databases or the Internet.

Because of this, it may seem that we don't need a way to express effects in Coq. After all, we can't make any use of them, right? Not at all. First, even though in Coq we can't interact with the outside world, we may want to model programs or programming languages that can. Second, remember that effects also encompass control mechanisms, and these are useful for our purposes. The function `label` we saw at the end of the previous chapter could look way better if expressed using the state effect.

Chapter 3

Design

Chapter 4

Examples

Chapter 5

A case study in proof engineering

Chapter 6

Conclusion

Bibliography

[1] Jeremy Gibbons and Ralf Hinze,
Just do It: Simple Monadic Equational Reasoning, 2011
http://www.cs.ox.ac.uk/jeremy.gibbons/publications/mr.pdf

[2] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom and Mike Hamburg,
Meltdown: Reading Kernel Memory from User Space, 2018
https://meltdownattack.com/meltdown.pdf

[3] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz and Yuval Yarom,
Spectre Attacks: Exploiting Speculative Execution, 2019
https://spectreattack.com/spectre.pdf

[4] https://heartbleed.com, 2019

[5] https://deepspec.org, 2019

[6] Gregory Travis,
How the Boeing 737 Max Disaster Looks to a Software Developer,
https://spectrum.ieee.org/aerospace/aviation/
how-the-boeing-737-max-disaster-looks-to-a-software-developer

[7] Georges Gonthier, *A computer-checked proof of the Four Colour Theorem*, 2005
https://www.cl.cam.ac.uk/~lp15/Pages/4colproof.pdf

[8] Georges Gonthier, *Formal Proof – The Four-Color Theorem*, 2008,
http://www.ams.org/notices/200811/tx081101382p.pdf

[9] Vladimir Voevodsky, *UNIVALENT FOUNDATIONS*, slides for a talk given at IAS on 26 March 2014,
http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_
files/2014_IAS.pdf

[10] https://coq.inria.fr/

[11] Morten Heine Sørensen, Paweł Urzyczyn,
Lectures on the Curry-Howard Isomorphism, 2006

[12] Benjamin C. Pierce, Andrew W. Appel and many others,
Software Foundations, 2019,
https://softwarefoundations.cis.upenn.edu/

[13] Yves Bertot and Pierre Castéran,
Interactive Theorem Proving and Program Development
Coq'Art: The Calculus of Inductive Constructions, 2004,
https://www.labri.fr/perso/casteran/CoqArt/

[14] Adam Chlipala, *Certified Programming with Dependent Types*, 2019, http://
adam.chlipala.net/cpdt/

[15] Tony Hoare,
Null References: The Billion Dollar Mistake,
https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Ho

[16] Saunders Mac Lane,
Categories for the Working Mathematician

[17] Brent Yorgey, *Abstraction, intuition, and the "monad tutorial fallacy"*,
https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-

[18] Eugenio Moggi, *Notions of computation and monads*,
https://person.dibris.unige.it/moggi-eugenio/ftp/ic91.pdf

[19] Mark P. Jones, *Functional Programming with Overloading and Higher-Order Polymorphism*,
http://web.cecs.pdx.edu/~mpj/pubs/springschool95.pdf

[20] Daan Leijen, *Koka: Programming with Row Polymorphic Effect Types*, 2014,
https://www.microsoft.com/en-us/research/wp-content/uploads/2016/
02/paper-20.pdf

[21] https://www.eff-lang.org/

[22] Sam Lindley, Connor McBride,
Do Be Do Be Do,
http://homepages.inf.ed.ac.uk/slindley/papers/
frankly-draft-march2014.pdf

[23] https://github.com/ocaml-multicore/ocaml-multicore

[24] https://bitbucket.org/pl-uwr/helium/src/master/