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

10 września 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* discuss its design and implementation and explain our approach to proof engineering (the formalized mathematics' equivalent of software engineering).
- In chapter 4 we demonstrate how to reason with the provided abstractions by proving some example effectful programs correct. We also prove a few general theorems and show that they may be useful for ordinary programmers.
- In chapter 5 we conclude by discussing related and future 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 judgment `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       cbn in H. 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

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.

2.1 Basic concepts

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 [22] 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, transformers and typeclasses

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 a rather annoying problem, because real-world code often requires way more than just two effects.

This deficiency can be remedied by introducing a more abstract design based on describing effects with **typeclasses** [19], which are basically the same thing as interfaces. Here each effect `E` is split into a specification and an implementation. The specification is a typeclass that expresses `E` by listing the operations that it supports. For example, the state effect supports `get` to retrieve the current state and `put` to modify the state. That a particular monad `M` supports the effect `E` is then expressed by implementing an instance of this typeclass for the monad `M`.

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 shared 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! Handlers 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.

Algebraic effects and handlers come in two flavours. One of them is to implement these abstractions at the library level using some new concepts like free monads or open unions, but also relying on the old menagerie of monads and monad transformers. This approach can be called `extensible-effects` after a Haskell library that proposed it. [20] [21]

The second flavour, the most recent and promising one, represented by languages such as Koka [22], Eff [23], Frank [24], Multicore OCaml [25] and Helium [26], instead aims to implement support for programming with algebraic effects and handlers at the language level, which has many practical advantages.

Native treatment of effects allows special treatment during compilation, which results in better performance. Integrating effect checking and effect inference with type checking and type inference results in better error messages and more manageable function signatures. Built-in support for effectful computations means lighter syntax and less boilerplate – for example, Haskell style `do`-notation is no longer needed, and built-in support for effect definitions means it is way easier to implement a custom effect than with `extensible-effects`, which requires one to dive a little into the library internals.

But probably the greatest advantage of this approach is its conceptual simplicity. Both `mtl` and `extensible-effects` suffer from a plethora of closely related concepts: monads and monad transformers, free and free-er monads, their operations and laws, the idea that they (indirectly) represent effects, and on top of it typeclasses, which are quite orthogonal to the subject at hand. This can be con-

trasted with the conceptual simplicity of native algebraic effects, with just the two core concepts of effects and handlers.

This conceptual simplicity is crucial for industry adoption. If the average Java programmer can't understand the basic ideas behind effects, there will be no wide adoption and thus no benefits from reduced number of bugs.

Chapter 3

Design

In this chapter we describe the design and implementation of *hsCoq*. We motivate the library's design by discussing possible ways of implementing effects in Coq, design choices related to abstraction and modularity mechanisms, various ways of using classes for formalizing concepts, and the matter of axioms.

3.1 Effects in Coq?

Now that we know the basics of effects, we can decide on how to best implement them in Coq.

First off, we need to recall that Coq implements Calculus of Inductive Constructions. This variant of type theory was not designed with the value-computation distinction in mind. Because of this, any native support of effects is ruled out – this would amount to implementing a whole new language, not to mention that such a language would have to be invented first - for example, at present no variant of type theory has both algebraic effects and dependent types.

Then we should ask ourselves, why do we want to have a way of expressing effects in Coq? After all, Coq is a proof assistant concerned mainly with theorem proving and formal verification that doesn't even support any kind of IO – you can't use it to write the simplest Hello World program, you can't connect to a database, you can't use it to write a web server (but see [27]).

We can answer this by recalling that effects are not only about communication with the outside world – they can also model control mechanisms like failure, exceptions or nondeterminism, and we surely want to make programming with these easier. The function `label` we saw at the end of chapter 1 could look way better expressed using the state effect.

Another reason is that, even though in Coq we can't interact with the outside world, we may want to model programs or programming languages that can. At

last, IO and other such effects may be added to Coq in the future and we want to be prepared for that (related languages, like Agda [28] and Idris [29] both have IO).

We are thus forced to implement our effects at the library level and because the `extensible-effects` approach depends crucially on many `mtl` abstractions as described earlier, we choose to implement the latter and leave the former for future work.

3.2 Typeclasses, canonical structures and modules

The good news is that Coq has various mechanisms for achieving abstraction and modularity, such as typeclasses, canonical structures and modules. The bad news is that these mechanisms are related, overlapping, and their use inside the standard library and in third party libraries is not very consistent. We will therefore try to explain the similarities and differences motivating our choice of using typeclasses to implement the library.

Typeclasses [37] [38] were originally invented for the purpose of overloading arithmetical operators in Haskell, but since then they were found to have many more applications and spread to other languages, like Rust.

In Coq, typeclasses are basically parametrized dependent records (collections of fields where types of later fields can depend on values of earlier fields and all of the fields can depend on the parameters) together with the mechanism of instance search (implemented using ordinary proof search) and a priority mechanism for deciding between overlapping instances. Listing 5 shows an example use of typeclasses to formalize a semigroup, a very simple algebraic structure that consists of a type together with an associative binary operation.

```
1 Class Semigroup (A : Type) : Type :=
2 {
3     op : A -> A -> A;
4     assoc : forall x y z : A, op (op x y) z = op x (op y z);
5 }.
```

Listing 5: Semigroups represented with typeclasses

Canonical structures are a mechanism unique to Coq. They were introduced in a 1998 PhD thesis in French [39] and remain quite obscure since then, but there were some attempts at explaining them better [40] [41] [42]. They are built on the foundation of parametrized dependent records, just like typeclasses, but they differ from the latter in the instance search mechanism – canonical structures are found using unification and not ad hoc proof search.

They also lack native handling of overlapping instances and backtracking, but this can be remedied with some hacks (see the conclusion section of [41] for a detailed comparison of typeclasses and canonical structures). They are usually coupled with the module mechanism – this approach is well present in the Mathematical Components library [43], a spin-off of the formalization of the Four Colour Theorem [7] [8]. Listing 6 shows a formalization of semigroups using canonical structures.

```
Module Semigroup.

    Record class (A : Type) := Class
    {
        op' : A -> A -> A;
        assoc : forall x y z : A, op' (op' x y) z = op' x (op' y z);
    }.

    Structure type := Pack
    {
        obj : Type;
        class_of : class obj;
    }.

    Definition op (e : type) : obj e -> obj e -> obj e :=
    let 'Pack _ (Class _ op' _) := e in op'.

    Arguments op {e} x y : simpl never.
    Arguments Class {A} op'.

End Semigroup.
```

Listing 6: Semigroups represented with canonical structures (and modules)

The last of Coq's abstraction mechanisms is the module system [44] [45], inspired by the module system of ML. A module, just like typeclasses and canonical structures, are very similar to parametrized dependent records (though not implemented using them), but they are less restrictive, because the body of a module can also contain vernacular commands, like inductive type definitions.

It is precisely these inductive type definitions that forbid Coq modules from being first-class (it may come as a surprise, but defining a new type can be considered an effect, because new named constants are added into the environment). Modules are also often used as namespaces, because all names declared in a module must be accessed by prefixing them with the module name, unless the module is imported into scope. Listing 7 shows a formalization of semigroups using modules (or, to be

more precise, module types).

```
Module Type Semigroup.

    Parameter A : Type.
    Parameter op : A -> A -> A.
    Axiom assoc :
        forall x y z : A, op (op x y) z = op x (op y z).

End Semigroup.
```

Listing 7: Semigroups represented with modules

Listings 5, 6 and 7 all look very much alike, except that the solution using canonical structures is longer, because one usually formalizes both the bundled and unbundled version of the concept (see next section for explanation).

Our abstraction and modularity mechanism of choice were typeclasses. The most important reason was that we wanted the library to be intelligible to people with Haskell background. An additional motivation was that the only other work on monads in Coq [30] uses canonical structures, so we wanted try something different.

3.3 Bundles, outlaws, setoids and other exotic animals

Choosing typeclasses as our abstraction mechanism doesn't exhaust the design space yet. In comparison to Haskell's, Coq's typeclasses are way more powerful, as they are both first-class and higher-order. For example, in Coq we can have functions which take functions between typeclasses as inputs and return functions between typeclasses as outputs, whereas in Haskell such constructions are not allowed. This particular example is a bit artificial, but when implementing a class hierarchy in Coq, there are at least three additional design choices that we can make which are not available in Haskell.

```
Class Semigroup : Type :=
{
    A : Type;
    op : A -> A -> A;
    assoc : forall x y z : A, op (op x y) z = op x (op y z);
}.
```

Listing 8: The bundled approach

```
Class Semigroup (A : Type) : Type :=
{
    op : A -> A -> A;
    assoc : forall x y z : A, op (op x y) z = op x (op y z);
}.
```

Listing 9: The unbundled approach

The first one I will call bundling. A bundled class is one that has no parameters – all the aspects of the concept being formalized are represented using fields. An example is shown in listing 8. An unbundled class is one where some of the fields were moved to become parameters instead, as shown in listing 9. A class can be unbundled to various degrees – in our example, we have only unbundled the type `A`, but we could have also unbundled the operation `op` (or even the law `assoc`, but that would be silly).

The bundled approach is better when we are interested in treating our concepts as objects on which we will operate. The unbundled approach is better when we want to use typeclasses as interfaces, which link the class parameters to its fields. Another advantage of the unbundled approach is that we can put an unbundled class together with its parameters in a new class (in a way very similar to that from listing 6) to recover the bundled version of this class. The other way around (recovering an unbundled class from its bundled version) is harder. We will therefore follow the unbundled approach.

```
Class Semigroup : Type :=
{
    A : Type;
    op : A -> A -> Type;
}.

Coercion A : Semigroup >-> Sortclass.

Class SemigroupLaws (S : Semigroup) : Prop :=
{
    assoc : forall x y z : S, op (op x y) z = op x (op y z);
}.

Class LawfulSemigroup : Type :=
{
    sgr : Semigroup;
    laws : SemigroupLaws sgr;
}.
```

Listing 10: The (bundled) lawless approach

Another approach to class design, whose example is shown in listing 10, I will call the lawful/lawless approach. The idea is a bit similar to bundling/unbundling: a lawful class is one that contains all the relevant laws, whereas in the lawless approach we split a class into two: the "main" class with all the data and operations and the "laws" class that contains all the (equational or other) laws that these data and operations should satisfy.

There are a few reasons why one might want to split laws from the main class. First, we may be interested in how fast the resulting code will execute in Coq (and not only after extraction), and carrying around proofs of all the laws and computing with them may slow down the execution speed. Second, we might want users of our classes to be able to program without having to prove anything at all. Third, we might be interested in proving some nontrivial properties, like equivalence of various classes (we will do this in chapter 4), in which case having to prove equality of proofs of the laws might be notoriously annoying or even require introducing an additional axiom – the Proof Irrelevance Axiom.

For *hsCoq* we adopt the lawful approach because we are interested not in programming per se, but in formally verified programming. In practice, it turns out that the lawful approach doesn't have a big impact on execution speed. Since we are interested in proving typeclass equivalences for practical purposes (to allow the programmer to implement what's easiest for him but get everything), we don't need

to do deal with proof equality in our proofs.

```
Class Setoid : Type :=
{
    carrier : Type;
    eqv : carrier -> carrier -> Prop;
    refl : forall x : carrier, eqv x x;
    sym : forall x y : carrier, eqv x y -> eqv y x;
    trans : forall x y z : carrier, eqv x y -> eqv y z -> eqv x z;
}.

Coercion carrier : Setoid >-> Sortclass.

Class Semigroup : Type :=
{
    A : Setoid;
    op : A -> A -> A;
    assoc : forall x y z : A, eqv (op (op x y) z) (op x (op y z));
    op_eqv : forall x x' y y' : A,
                eqv x x' -> eqv y y' -> eqv (op x y) (op x' y');
}.
```

Listing 11: The (bundled) setoid approach

The last thing we could do is using setoids instead of types to formalize the concepts of interest. A setoid is a type together with an equivalence relation and this approach boils down to replacing all types wiht setoids and adding the necessary compatibility laws for operations, as shown in listing 11.

The main use of setoids is in dealing with equality – because we cannot form quotient types in Coq, any kind of mathematics that requires quotients (algebra), gluing (topology), or just simply changing the notion of equality for a particular structure, needs to be carried out using setoids. We would need to use them if we wanted to focus on the category-theoretical aspect of monads and monad transformers, but since we are interested mainly in the practical aspects, we will not need any setoids.

To sum it up, bundled classes are best used for formalizing mathematics, with setoids when in need of quotients/gluing, whereas unbundled classes are best used for representing interfaces, possibly without the laws if we care only about programming and not proving.

3.3.1 The final class hierarchy

The choices we made mostly exhaust the design space. All other basic basic design choices (the class hierarchy up to monad transformers, naming of modules and functions, etc.) were made to match Haskell. Together with the monadic effect classes taken from [1], this results in the hierarchy of classes and their instances shown in figure 3.1.

3.4 Axioms

A very important property of every Coq development is being axiom free. The main reason is that the computational content of type theory is very axiom-sensitive. Because every axiom is just an element of some type, postulating it breaks the important property of canonicity. For example, if we postulate an axiom `nimber` of type `nat`, how does the term `nimber + 42` evaluate? The answer is that we don't know (or rather, it doesn't), because the definition of addition only handles the cases of zero and successor. This term is therefore stuck, i.e. it "doesn't compute".

hsCoq doesn't have the property of being axiom free. This is because a lot of monad laws (and other laws too), to be proved, require the Functional Extensionality Axiom, which states that two functions are equal as soon as they have equal results for all inputs. We therefore adopt this axiom from the very beginning and use it wherever we need it without explicit mention. The resulting loss of canonicity isn't that bad, however, because it only prevents us from performing some computations with equality proofs, which are not at all important. We therefore consider adopting this axiom to be justified.

3.5 Proof engineering

Now that we know our axioms, we can start proving, right? Well, not really. When programmers want to create something bigger than a toy program, they face the question of software engineering – what architecture will be the best, what design will be the most robust, which tools to use etc. Writing assembly by hand or non-structured programming are not considered good software engineering practices anymore.

The same is true of mathematicians who go formal – when they want to prove lots of theorems that form a coherent theory, they face the question of proof engineering – what "architecture" will express best the structure of the theory, what style of proving will let them prove their theorems in the most painless way and so on. Writing proofterms by hand or simple proofscripts composed only of the most basic tactics are not good proof engineering practices. We should therefore stick

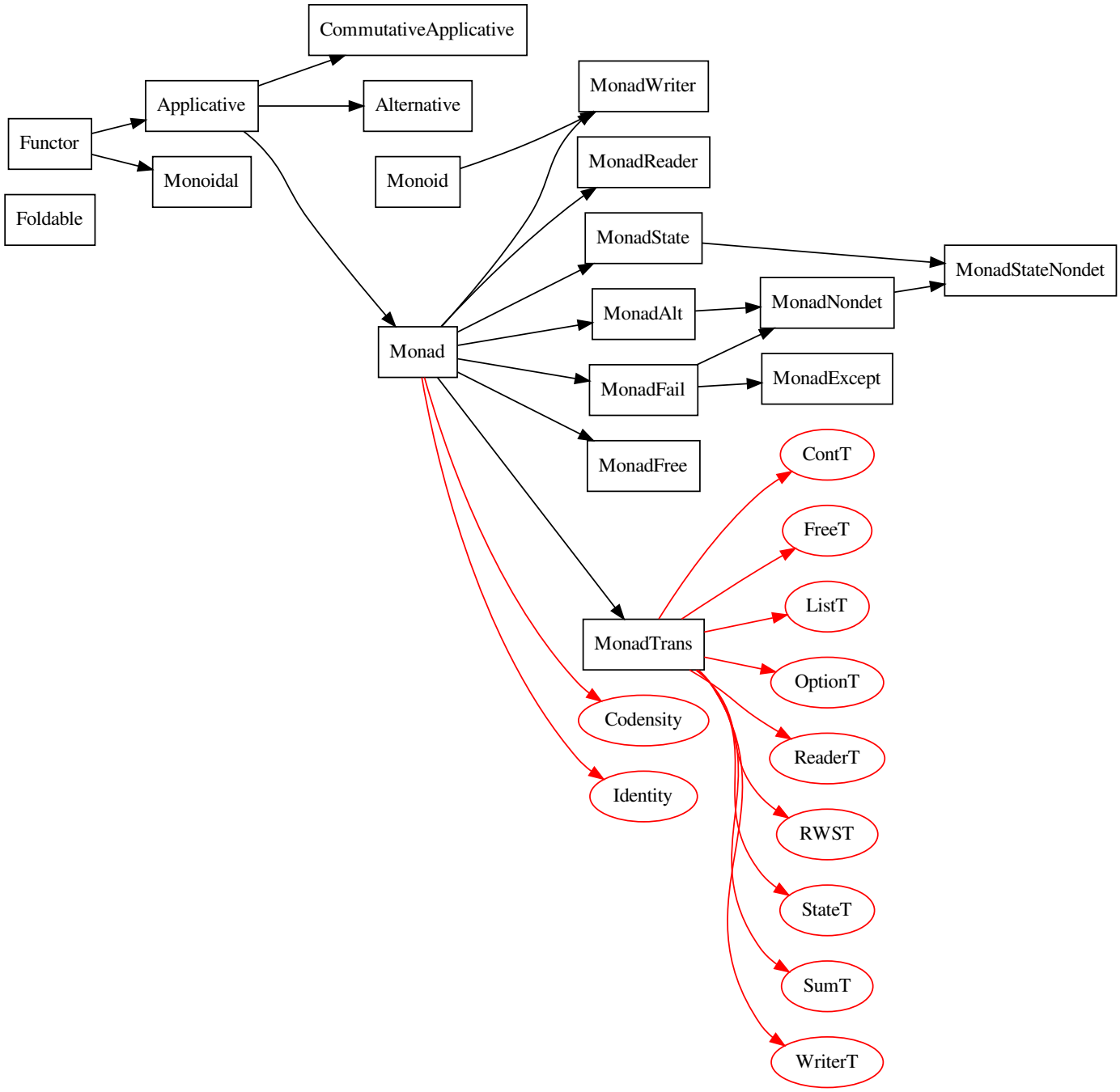

Figure 3.1: The library structure. Black boxes are classes, a black arrow from A to B means that class B depends on class A. Red ovals are class instances and a red arrow from A to B means that B is an instance of class A. A few classes were omitted (mainly variants of `MonadAlt`) and instances of the monadic effect classes (`MonadWriter` through `MonadFree`) were also omitted, as they would make the diagram unreadable.

with some good practices, but what are they?

Ltac, Coq's tactic language, is a very rich language and this has led to many proof styles being employed in various places. The most important distinction is between manual proofs (which consist of a series of tactics that prove the theorem step by step, like in the example in chapter 1) and automated proofs (which usually take the form of a big tactic that proves a whole theorem all at once).

Another distinction is between reflective and non-reflective proofs. Reflective proofs are proofs written in Coq's term language that operate on syntactic representations of theorems. The sole role of Ltac in this approach is reification, i.e. transforming these theorems into their syntactic representations (this is something that Coq's term language cannot do, because it would result in inconsistency). Non-reflective proofs are the ordinary Ltac-based proofs that don't use the above machinery of reflection and reification. This gives rise to four proof styles:

- Manual, non-reflective: often used by beginners and not considered a good practice, unless we're dealing with hard theorems that don't automate well.
- Manual, reflective: this is the style most associated with the SSReflect proof language [56] [57], used in the formalization of the Four Colour Theorem [7] [8] and the Mathematical Components library [43].
- Automated, non-reflective: often used by experts for theorems with lots of repetitive cases or collections of similar theorems. An example is the field of programming language formalization, where proofs can have hundreds of cases, most of them utterly obring. A very known proponent of this style is Adam Chlipala, who advocates it in his book [14].
- Automated, reflective: most often used for solving specific goals of a specific kind, like Presburger arithematic (the arithmetic of natural numbers with addition and multiplication by a constant – Coq's tactic `omega`) or simplifying expressions and proving equalities in rings (Coq's tactic `ring`).

In our case, there will be a massive amount of typeclass instances that have to be proven to satisfy the same laws over and over again. Almost all of these proofs are very repetitive and predictable. They amount mostly to unfolding definitions, rewriting equational laws, decomposing hypotheses and implicational reasoning. This means we want our proofs to be automated and not manual.

It may also seem that we want the tactics to be reflective, because the typeclasses we are dealing with are very well structured mathematically. This is however not the case, because we want our tactics to also be of use to users of our library, but we cannot predict what kinds of monads and effect they will want to create. This makes makes the world of `Functor`s, `Applicative`s, `Monad`s and effects not a closed world, like that of Presburger arithmetic or ring expressions, but an open one. We therefore choose to follow the automated non-reflective approach.

Chapter 4

Implementation, theorems and examples

In this chapter we describe the implementation of our library, including some subtleties that we didn't foresee coming during design, explain our approach to proof automation (how to prove all these laws without too much labor), give some example use cases of our library and prove a few theorems which may seem unnecessary at first, but turn out to have some practical use.

4.1 The core library

```
8  Class Functor (F : Type -> Type) : Type :=
9  {
10     fmap : forall {A B : Type}, (A -> B) -> (F A -> F B);
11     fmap_id :
12       forall A : Type, fmap (@id A) = id;
13     fmap_comp :
14       forall (A B C : Type) (f : A -> B) (g : B -> C),
15         fmap (f .> g) = fmap f .> fmap g;
16 }.
```

Listing 12: The `Functor` typeclass

At the root of the library are, of course, functors, shown in listing 12. There should be nothing surprising to people who know them from Haskell, maybe except that the laws are now part of the definition. Note that we use `.>` to denote forward function composition, so that `f .> g` means `fun x => g (f x)`

A few category-theoretical remarks: unlike Haskell types and functions, which do *not* form a category (see [46] for details), Coq types and functions do form a category (or to be more precise, we can form a category whose objects are types from a given universe and morphisms are functions between them). The `Functor` typeclass represents endofunctors in this category. Note that all such functors `F` are strong in the sense that there is a morphism `strength : A -> F B -> F (A * B)`.

```
Class Applicative (F : Type -> Type) : Type :=
{
    is_functor :> Functor F;
    pure : forall {A : Type}, A -> F A;
    ap : forall {A B : Type}, F (A -> B) -> F A -> F B;
    identity :
      forall (A : Type) (ax : F A), ap (pure id) ax = ax;
    composition :
      forall
        (A B C : Type)
        (f : F (A -> B)) (g : F (B  -> C)) (x : F A),
          ap (ap (ap (pure compose) g) f) x = ap g (ap f x);
    homomorphism :
      forall (A B : Type) (f : A -> B) (x : A),
        ap (pure f) (pure x) = pure (f x);
    interchange :
      forall (A B : Type) (f : F (A -> B)) (x : A),
        ap f (pure x) = ap (pure (fun f => f x)) f;
    fmap_pure_ap :
      forall (A B : Type) (f : A -> B) (x : F A),
        fmap f x = ap (pure f) x;
}.

Coercion is_functor : Applicative >-> Functor.
```

Listing 13: The `Applicative` typeclass

The next step in the typeclass hierarchy are `Applicative` functors, shown in listing 13. From the definition we see that an `Applicative` functor is a `Functor` into which we can inject a value (the function `pure`) and which allows to apply a function wrapped in the functor to an argument wrapped in the functor (the function `ap`, which is short for "apply").

These two operations, `pure` and `ap`, have to satisfy some laws and this is the first

time that some subtle complications arise. The four laws that are usually presented (such as in [47]) are `identity`, `composition`, `homomorphism` and `interchange`. Other sources (like [48] and [49]) also mention a fifth law, `fmap_pure_ap`, but they often either state that this law follows from the other laws using naturality or they keep quiet about it.

In Coq naturality is not internalized (and in fact it can be broken by assuming classical axioms like the Law of Excludded Middle), so this law does not follow from the others. A more intuitive reason is that the other four laws don't mention any `Functor` concept at all, so there's no way to link `fmap` with `pure` and `ap` in any way. This is why we include this law in our definition.

This, however, leads to some further troubles, because it turns out that this set of five laws is not minimal. For example, `identity` follows from `fmap_pure_ap` and `fmap_id` (for details, see the file `Theory/Laws/ApplicativeLaws.v`). It could lead us to drop `identity` from the definition of `Applicative`, but we decided not to.

```
Definition reassoc
  {A B C : Type} : (A * B) * C -> A * (B * C) :=
    fun '((a, b), c) => (a, (b, c)).

Definition par
  {A A' B B' : Type} (f : A -> B) (g : A' -> B') : A * A' -> B * B' :=
    fun '(a, b) => (f a, g b).

Notation "f *** g" := (par f g) (at level 40).

Class Monoidal (F : Type -> Type) : Type :=
{
    is_functor :> Functor F;
    default : F unit;
    pairF : forall {A B : Type}, F A -> F B -> F (A * B)%type;
    pairF_default_l :
      forall (A : Type) (v : F A),
        fmap snd (pairF default v) = v;
    pairF_default_r :
      forall (A : Type) (v : F A),
        fmap fst (pairF v default) = v;
    pairF_assoc :
      forall (A B C : Type) (a : F A) (b : F B) (c : F C),
        fmap reassoc (pairF (pairF a b) c) = pairF a (pairF b c);
    natural :
      forall
        (A A' B B' : Type)
        (f : A -> A') (g : B -> B')
        (a : F A) (b : F B),
          fmap (f *** g) (pairF a b) = pairF (fmap f a) (fmap g b);
}.
```

Listing 14: The `Monoidal` typeclass

Another question we could ask is: what are `Applicative` functors categorically? It is a sad question, because it gets answered rarely, but the answer is actually very simple – `Applicative` functors are just `Monoidal` functors in disguise! The definition of a `Monoidal` functor is shown in listing 14.

The category of Coq types and functions has a terminal object (the `unit` type) and binary products, which together (with some coherence conditions too) form its monoidal structure. A `Monoidal` functor (and thus, an `Applicative` functor) is a

`Functor` which preserves this monoidal structure in the sense that `default`, `pairF`, `fmap fst`, `fmap snd` and `fmap reassoc` behave just like `tt` (the only value of the `unit` type), `pair`, `fst`, `snd` and `reassoc` and they do this in a natural way.

We have formally proved that `Applicative` and `Monoidal` are equivalent (see the file `Theory/Applicative_is_Monoidal.v`), which justifies the above explanation of the categorical semantics of `Applicative`. It also assures us that our set of `Applicative` laws is correct and that putting `fmap_pure_ap` in the definition was necessary.

But there is yet another perk of this proof, which shows that proving is not only of theoretical interest, but also has practical aspects – thanks to this proof users of our library can now define an instance of either `Applicative` or `Monoidal` and get an instance of the other typeclass for free, something which is not possible in a language like Haskell.

```
21 Class Monad (M : Type -> Type) : Type :=
22 {
23     is_applicative :> Applicative M;
24     bind : forall {A B : Type}, M A -> (A -> M B) -> M B;
25     bind_pure_l :
26       forall (A B : Type) (f : A -> M B) (a : A),
27         bind (pure a) f = f a;
28     bind_pure_r :
29       forall (A : Type) (ma : M A),
30         bind ma pure = ma;
31     bind_assoc :
32       forall (A B C : Type) (ma : M A) (f : A -> M B) (g : B -> M C),
33         bind (bind ma f) g = bind ma (fun x => bind (f x) g);
34     bind_ap :
35       forall (A B : Type) (mf : M (A -> B)) (mx : M A),
36         mf <*> mx = bind mf (fun f => bind mx (fun x => pure (f x)));
37 }.
38
39 Coercion is_applicative : Monad >-> Applicative.
```

Listing 15: The `Monad` typeclass

Next come monads, shown in listing 15. The definition is very similar to modern Haskell, where `Applicative` is a superclass of `Monad` since GHC 7.10.1. The three usual laws are there, but there's also a fourth one, namely `bind_ap` (this law is also mentioned in [50]).

The reason is that in the category of Coq types and functions each `Monad` is monoidal (in fact, the `Monad` typeclass represents a strong monoidal monad on the category of Coq types and functions), which means that it induces an instance of `Applicative`. However, it does so in two nonequivalent ways, which correspond to the two ways of defining `ap` using `bind` (which is sequential). To avoid potential doubling of `Monad` instances, we put the law `bind_ap` in the definition, which forces `ap` to evaluate its arguments from left to right.

Just as in the `Applicative` case, an additional trouble is that this set of four laws is not minimal. For example, `bind_pure_l` follows from `fmap_id`, `fmap_pure_ap`, `bind_ap` and `bind_pure_l` – see the file `Theory/Laws/MonadLaws.v` for details. We could have therefore dropped `bind_pure_l` from the definition, but we didn't decide to do so.

To make sure that this definition of monads makes sense, we proved that it is equivalent to a simpler one, which uses just `pure`, `bind` and the appropriate laws. We also prove it equivalent to a definition based on `Applicative`, but with `join` instead of `bind`. Of course these two equivalences can be used by the user to define an instance of whichever kind of `Monad` is the easiest for him and get the other ones for free.

Another interesting fact is that we also tried to prove this definition equivalent to one based on `pure` and `compM` (forward composition of monadic functions), which is closely related to the definition of a monad's Kleisli category, but we failed. See the directory `Theory/Equivs` for more details.

```
60 Module MonadNotations.
61
62 Notation "mx >>= f" := (bind mx f) (at level 40).
63 Notation "f >=> g" := (compM f g) (at level 40).
64
65 Notation "x '<-' e1 ; e2" := (bind e1 (fun x => e2))
66   (right associativity, at level 42, only parsing).
67
68 Notation "e1 ;; e2" := (e1 >> e2)
69   (right associativity, at level 42, only parsing).
70
71 Notation "'do' e" := e (at level 50, only parsing).
72
73 End MonadNotations.
```

Listing 16: Monad notations

But what would monads be without their do-notation? As it turns out, we can define it using Coq's powerful notation mechanism, as shown in listing 16. This do-notation is not perfect, because we need to follow each single `bind` with a semicolon and each use of `>>` (the operation that runs two computations in sequence, throwing away the result of the first) with a double semicolon. Also, the "do" part can be skipped without consequence, because it doesn't do anything. Despite these little annoyances, this do-notation works very well in practice.

It is also worth mentioning that our library contains all the usual utility functions associated with `Applicative` and `Monad`. In Haskell, these functions are often duplicated, with the `Applicative` version having the suffix `A` and the `Monad` version having the suffix `M`. In *hsCoq* these functions are deduplicated, i.e. there is always only one version of each, implemented using the weakest typeclass that suffices for the definition. See the files `Control/Applicative.v` and `Control/Monad.v` for their definitions.

```
13 Definition Free (F : Type -> Type) (A : Type) : Type :=
14   forall X : Type, (A -> X) -> (F X -> X) -> X.
15
16 Section Free.
17
18 Variable F : Type -> Type.
19
20 Definition fmap_Free
21   {A B : Type} (f : A -> B) (x : Free F A) : Free F B :=
22     fun X pure wrap => x X (f .> pure) wrap.
23
24 Instance Functor_Free : Functor (Free F) :=
25 {
26     fmap := @fmap_Free
27 }.
28 Proof. all: reflexivity. Defined.
```

Listing 17: The `Free` monad

We have encountered problems that are not present in Haskell not only with the definitions of `Applicative`s and `Monad`s, but also when defining their instances. Coq, being a theorem prover, requires all inductive definitions to be strictly positive (i.e. the type being defined cannot stand to the left of an arrow or be applied to an opaque type-level function). This prevents us from defining the `Free` monad in the standard way, the problem being that the occurrence of `Free F A` is not strictly positive in `F (Free F A)`.

```
Inductive Free (F : Type -> Type) (A : Type) : Type :=
| Pure : A -> Free F A
| Wrap : F (Free F A) -> Free F A.
```

Our solution to this problem was to use Church encoding instead of inductive types. The resulting definition is shown in listing 17. In this definition, we quantify over a type `X`, which represents the type we're defining, and then we put all the constructors into one big dependent function whose result is `X`.

This definition is fine in the sense that it allows us to define all necessary functions are prove all the laws. This may seem surprising, because it is well known that Church-encoded types don't support induction (i.e. their induction principles are not definable in Coq) [51]. What's even more surprising is that they satisfy all these laws definitionally, i.e. we can prove all of them using just the tactic `reflexivity` – see the proof of `Functor` laws in listing 17.

Of course `Free` is not the only `Monad` instance. We provide all the standard ones: `Cont`, `Free`, `Identity`, `list`, `option`, `Reader`, `RWS`, `State`, `Sum`, `Writer` and there are also some less standard ones: `Codensity` (see [52]), `Lazy` (which attempts to capture lazy evaluation in a monad) and `RoseTree` (a tree which keeps values only in its leaves).

```
Class MonadTrans (T : (Type -> Type) -> Type -> Type) : Type :=
{
    is_monad : forall (M : Type -> Type), Monad M -> Monad (T M);
    lift :
      forall {M : Type -> Type} {_inst : Monad M} {A : Type},
        M A -> T M A;
    lift_pure :
      forall (M : Type -> Type) {_inst : Monad M} (A : Type) (x : A),
        lift (pure x) = pure x;
    lift_bind :
      forall
        {M : Type -> Type} {_inst : Monad M} (A B : Type)
        (x : M A) (f : A -> M B),
          lift (x >>= f) = lift x >>= (f .> lift)
}.
```

Listing 18: The `MonadTrans` typeclass

The pinnacle of the core library are monad transformers, represented by the

class `MonadTrans`, as shown in listing 18. As we can see from the definition, a monad transformer is something which takes a monad and returns a monad, together with an operation `lift` that turns a base monad computation into a transformed monad computation, together with some obvious laws: `lift`ing a pure value has to result in a pure value, and `lift` it has to commute with `bind`.

Luckily, monad transformers are just as unproblematic as `Functor`s were. The library implements all the standard instances, including `ContT`, `FreeT`, `ListT`, `OptionT`, `ReaderT`, `RWST`, `StateT`, `SumT` and `WriterT`.

```
12 Definition ListT
13   (M : Type -> Type) (A : Type) : Type :=
14     forall X : Type, M X -> (A -> M X -> M X) -> M X.
15
16 Module ListT_Notations.
17
18 Notation "[[ ]]" :=
19   (fun X nil _ => nil).
20 Notation "[[ x ]]" :=
21   (fun X nil cons => cons x nil).
22 Notation "[[ x ; y ; .. ; z ]]" :=
23   (fun X nil cons => cons x (cons y .. (cons z nil) ..)).
24
25 End ListT_Notations.
```

Listing 19: ListT monad transformer

The most noteworthy of these instances is `ListT`. Haskell's version of this monad transformer is broken (it doesn't respect the laws) and numerous solutions were proposed [53] [54] [55]. Our definition, shown in listing 19, follows [53] most closely, but once again we had to use Church encoding because the inductive version would fail the strict positivity criterion.

Thanks to Church encoding, all proofs (besides those that refer to the underlying monad) go through by `reflexivity` and this is a very strong sign that our `ListT` is really done right. Also thanks to Coq's notation mechanism, we can write `ListT` computations (`[[x; y; z]]`) nearly the same as we do ordinary lists (`[x; y; z]`).

4.2 Proof engineering

Are we so lucky that all proofs whatsoever go through just by `reflexivity`, like for the Church-encoded monads and monad transformers? Of course not – most proofs

require a bit more than just `reflexivity`.

This bit is surprisingly small – the core of our automation is shown in listing 20. The first ten lines are tactics for conveniently reasoning by functional extensionality (so that we can write `exts` instead of `extensionality a; extensionality b; ...`). Then comes the tactic `unmatch`, which destructs the innermost subject of a pattern match, and the tactic `destr`, which deconstructs products and units, and case splits on suns and subjects of pattern matching. `simplify` is a tactic that introduces variables to context, reasons by functional extensionality when it's possible and then calls `destr` to further simplify the goal.

`hs` is the main workhorse tactic of the library – after introducing variables to the context, it proceeds by repeatedly unfolding definitions and rewriting equations and then tries to finish the goal off by unfolding a few more definitions and reasoning by `reflexivity` and `congruence` (which is an implementation of the congruence closure algorithm). At last, the `monad` tactic tries to simplify the goal, solve it using `hs` and if that isn't enough, then it tries some tricks for proving equations between monadic computations.

How does the rewriting and unfolding exactly work? It is quite simple: the tactics `autorewrite` and `autounfold` are called with some **hint databases** as arguments. For `autorewrite`, the hint database contains names of equations and for `autounfold` it contains names of defined objects. These tactics work by repeatedly rewriting/unfolding these equations/definitions in some order (determined by their implementations) until no more rewrites/unfoldings are possible.

Where do the hint databases come from? We have to create them and this is what line 29 in listing 20 does: it declares that the theorems named `id_eq`, `id_left`, `id_right` belong to the hint database named `HSLib`. Each time we encounter some useful equations (either in a typeclass definition or after proving them by hand), we add them to this hint database and from that point on `autorewrite` will be able to use them. We have to do the same for `autounfold`: each time we define something we will want to get unfolded, we add it to a hint database named `HSLib` and from that point on `autounfold` will be able to unfold it for us.

Of course, these automation tactics are not enough to deal with all instances of all typeclasse. In a few cases we have to finish the proofs by hand. In a few other cases these tactics don't help us at all – the most obvious example are lists, which require inductive proofs, but our tactics don't do any induction. The `Applicative` instance for lists is probably the most automation-resilient proof of the whole library. There are also a few proofs to which these tactics weren't even meant to apply, like the proof of equivalence of `Applicative` and `Monoidal` functors, which requires custom ad-hoc automation. Nonetheless, it's still surprising that some fifty lines of Ltac code can prove so many useful theorems.

```
Require Export Coq.Logic.FunctionalExtensionality.

Tactic Notation "ext" ident(x) := extensionality x.
Tactic Notation "ext2" ident(x) ident(y) := ext x; ext y.
Tactic Notation "ext3" ident(x) ident(y) ident(z) :=
  ext x; ext y; ext z.
Tactic Notation "ext" := let x := fresh "x" in ext x.
Ltac exts := repeat ext.

Ltac unmatch x :=
match x with
    | context [match ?y with _ => _ end] => unmatch y
    | _ => destruct x
end.

Ltac destr := repeat
match goal with
    | x : _ * _ |- _ => destruct x
    | x : _ + _ |- _ => destruct x
    | x : unit |- _ => destruct x
    | |- context [match ?x with _ => _ end] => unmatch x
end.

Ltac simplify :=
  cbn; intros; exts; destr.

Hint Rewrite @id_eq @id_left @id_right : HSLib.

Ltac hs :=
  cbn; intros;
  repeat (autorewrite with HSLib + autounfold with HSLib);
  try (unfold compose, id, const; congruence + reflexivity).

Ltac monad := repeat (simplify; try (hs; fail);
match goal with
    | |- ?x >>= _ = ?x =>
        rewrite <- bind_pure_r; f_equal
    | |- ?x = ?x >>= _ =>
        rewrite <- bind_pure_r at 1; f_equal
    | |- ?x >>= _ = ?x >>= _ => f_equal
    | _ => hs
end).
```

Listing 20: The automation tactics

4.3 Typeclass-based effects

We now describe "typeclass-based effects" (note that this is not a widely used term and, to be honest, we don't know if there is any widely used name for this approach to effects). Their popularity in Haskell is due to the `mtl` library (after which we also called this approach "the `mtl` approach"), which was originally inspired by [19]. Equational reasoning with such typeclass-based effects was first studied in [1]. This thesis is, to our knowledge, one of the earliest attempts to formalize this approach in Coq (with only [30] being published earlier).

```
6  Class MonadFail (M : Type -> Type) (inst : Monad M) : Type :=
7  {
8      fail : forall {A :  Type}, M A;
9      bind_fail_l :
10       forall (A B : Type) (f : A -> M B),
11         fail >>= f = fail;
12 }.
```

Listing 21: The `MonadFail` typeclass

In practice the whole thing works like this: each effect gets its own typeclass that lists all operations related to that effect and also equational laws that relate these operations to each other and to `pure` and `bind`. If a monad implements such a typeclass, it means that this monad supports that effect, i.e. it can be used to run programs that use that effect.

An example is shown in listing 21. This is one of the simplest possible effects – failure. A monad that supports failure should provide an operation `fail` that is related to `bind` by the law `bind_fail_l`, which intuitively says that all computations following a failure also fail.

```
Class MonadAlt (M : Type -> Type) (inst : Monad M) : Type :=
{
    choose : forall {A : Type}, M A -> M A -> M A;
    choose_assoc :
      forall {X : Type} (a b c : M X),
        choose (choose a b) c = choose a (choose b c);
    bind_choose_l :
      forall (A B : Type) (x y : M A) (f : A -> M B),
        choose x y >>= f = choose (x >>= f) (y >>= f);
}.
```

Listing 22: The `MonadAlt` typeclass

Another example, shown in listing 22, is the `MonadAlt` typeclass, which represents the nondeterministic choice effect. A monad that supports this effect should provide an operation `choose` that is associative (the law `choose_assoc`, which intuitively means that the various ways of associating a multiple choice don't make the choice biased) and left-distributes over `bind` (the law `bind_choose_l`, which intuitively means that a computation that depends on a choice is the same as a choice between computations).

```
Class MonadNondet (M : Type -> Type) (inst : Monad M) : Type :=
{
    instF :> MonadFail M inst;
    instA :> MonadAlt M inst;
    choose_fail_l :
      forall (A : Type) (x : M A),
        choose fail x = x;
    choose_fail_r :
      forall (A : Type) (x : M A),
        choose x fail = x;
}.

Coercion instF : MonadNondet >-> MonadFail.
Coercion instA : MonadNondet >-> MonadAlt.
```

Listing 23: The `MonadNondet` typeclass

A great strength of typeclass-based effects is that we can easily create new effects by composing previously defined effects and linking their operations with additional

laws. An example is shown in listing 23. The typeclass `MonadNondet` represents nondeterminism, understood as the ability to fail or make a binary choice (note that this is equivalent to being able to choose from a finite number of alternatives). The "failure" part comes from the `MonadFail` typeclass and the "choice" part comes from the `MonadAlt` typeclass. Moreover, there are two additional laws (`choose_fail_l` and `choose_fail_r`) which intuitively mean that we don't choose failure if we are not forced to.

```
Class MonadReader
  (R : Type) (M : Type -> Type) (inst : Monad M) : Type :=
{
    ask : M R;
    ask_ask : ask >> ask = ask;
}.
```

Listing 24: The `MonadReader` typeclass

```
Instance MonadReader_MonadTrans
  (T : (Type -> Type) -> Type -> Type) (instT : MonadTrans T)
  (M : Type -> Type) (instM : Monad M)
  (R : Type) (instMR : MonadReader R M instM)
  : MonadReader R (T M) (is_monad _ instM) :=
{
    ask := lift ask
}.
Proof.
  rewrite lift_constrA, ask_ask. reflexivity.
Defined.
```

Listing 25: The `MonadReader` instance for all monad transformers can be defined at once

But typeclass-based effects also have a great weakness which was already mentioned – they require a quadratic amount of instances to be defined (in `mtl` the formula is the number of typeclasses times the number of monad transformers). In *hsCoq* the situation is very similar, but surprisingly, there is a single exception – the `MonadReader` typeclass, shown in listing 24.

It represents a very simple effect that lets us `ask` the environment for a value of type `R`. The law `ask_ask` states that asking twice is the same as asking just once. Because this typeclass is so simple, it turns out we can define a `MonadReader`

instance for all monad transformers in one fell swoop, something that is not possible in Haskell. This reduces the amount of boilerplate we have to write, even though the amount of instances is still quadratic, and gives some hope that maybe this will be possible for other kinds of effect typeclasses. This hope is in vain however, as we empirically checked that this trick doesn't work even for other very simple effects, like `MonadFail`.

We have implemented most typeclasses that appear in [1], including `MonadExcept`, `MonadFail`, `MonadAlt` (and its variants `MonadAltBag`, `MonadAltSet` and `MonadAlt2`), `MonadNondet`, `MonadState`, `MonadStateNondet`, but we omitted the probability monad. We have also formalized most of [1] – see the file `Theory/JustDoIt.v` for details.

We have also implemented a few effect typeclasses that should be familiar from `mtl`, like `MonadReader`, `MonadWriter` and `MonadFree`, but there are some differences: as we have seen, `MonadReader` lacks the `local` operation that is present in `mtl` and it was quite hard to figure out the "correct" laws that `MonadWriter` instances should obey (what we did was empirically test which laws were satisfied by most instances and adopted these).

```
14 Class MonadState
15   (S : Type) (M : Type -> Type) (inst : Monad M) : Type :=
16 {
17     get : M S;
18     put : S -> M unit;
19     put_put :
20       forall s1 s2 : S, put s1 >> put s2 = put s2;
21     put_get :
22       forall s : S, put s >> get = put s >> pure s;
23     get_put :
24       get >>= put = pure tt;
25     get_get :
26       forall (A : Type) (k : S -> S -> M A),
27         get >>= (fun s : S => get >>= k s) =
28         get >>= fun s : S => k s s
29 }.
```

Listing 26: The `MonadState` typeclass

4.4 Examples

We finish this chapter with two examples of reasoning with our monads and effects. First we revisit the example from chapter 1 and see how it can be improved with do-notation and automation tactics. We finish with a formalization of the fast product example from [1].

4.4.1 Tree labeling, again

The tree labeling example is shown in listing 27. The definitions of `Tree` and `size` are as before, so they are omitted. We want to implement the `label` function using the state effect, so we start by importing the `MonadState` typeclass, shown in listing 26. The state effect supports two operations, `get` and `put`, which are related by four very intuitive laws – we encourage the reader to figure out that they mean.

The `label` function looks very different than before. Its type is more complicated now, because we have generalized it to work for any monad supporting the `MonadState` effect. However, the body of the definition is much easier to read because of do-notation.

The function works just like before. If we hit a `Leaf`, we retrieve the state of the counter and use it to label the `Leaf`. If we hit a `Node`, we label the left subtree, increase the counter by one (`modify S` means "modify the state by applying the function `S`", which is the successor function), label the right subtree and then we join these two labeled subtrees with `Node`.

To get the function we really want (`lbl`, which starts labeling from zero) we have to instantiate the definition of `label` with some monad supporting the state effect and we of course choose the `State` monad. We see that we can compute with this definition just like we could with the old one and the result is, of course, the same.

The correctness theorem (which states that after running `label` on a tree `t`, the counter increases by the size of `t`) looks quite similar the old one, just as does the proof. There are two important things to notice. First, we prove this theorem only for the concrete `State` monad and not for any monad supporting the `MonadState` effect. This is because the state effect doesn't give us any way of pulling the state outside the monad and we need this ability to even state the theorem. Second, we crucially use out automation tactic `hs`, which change the proof state from barely readable (it mentions the particular definitions of `pure`, `bind` and so on for the `State` monad) to one that strongly resembles the earlier, non-monadic version from chapter 1.

The overall result is that the `label` function is more general, its definition is easier to read and the proof didn't get any harder. Not bad.

```
Require Import Control.Monad.Class.MonadState.

Fixpoint label
  {M : Type -> Type} {inst : Monad M} {instS : MonadState nat M inst}
  {A : Type} (t : Tree A) : M (Tree (A * nat)) :=
match t with
    | Leaf x => do
        n <- get;
        pure $ Leaf (x, n)
    | Node l r => do
        l' <- label l;
        modify S;;
        r' <- label r;
        pure $ Node l' r'
end.

Require Import Control.Monad.State.

Definition lbl {A : Type} (t : Tree A) : Tree (A * nat) :=
  fst (@label (State nat) _ _ _ t 0).

Compute lbl (Node (Node (Leaf true) (Leaf true)) (Leaf false)).
(* = Node (Node (Leaf (true, 0)) (Leaf (true, 1))) (Leaf (false, 2))
   : Tree (bool * nat) *)

Theorem label_size :
  forall (A : Type) (t : Tree A) (n : nat),
    1 + snd (@label (State nat) _ _ _ t n) = n + size t.
Proof.
  induction t as [| l IHl r IHr]; hs.
    rewrite <- plus_comm. reflexivity.
    case_eq (label l n); intros t1 m1 H1.
      case_eq (label r (S m1)); intros t2 m2 H2.
      specialize (IHl n). specialize (IHr (S m1)).
      rewrite H1, H2 in *. cbn in *.
      rewrite IHr, <- plus_Sn_m, IHl, !plus_assoc. reflexivity.
Qed.
```

Listing 27: The tree labeling example from chapter 1, generalized with the `MonadState` typeclass and improved with do-notation

4.4.2 Fast product

A fast product is a procedure for calculating the product of a list that returns zero as soon as it encounters a zero on this list, without having to consider the rest of the list. We will implement it using exceptions, so first we have to introduce the `MonadExcept` effect, shown in listing 28.

```
14 Class MonadExcept
15   (M : Type -> Type) (inst : Monad M)  : Type :=
16 {
17     instF :> MonadFail M inst;
18     catch : forall {A : Type}, M A -> M A -> M A;
19     catch_fail_l :
20       forall (A : Type) (x : M A),
21         catch fail x = x;
22     catch_fail_r :
23       forall (A : Type) (x : M A),
24         catch x fail = x;
25     catch_assoc :
26       forall (A : Type) (x y z : M A),
27         catch (catch x y) z = catch x (catch y z);
28     catch_pure :
29       forall (A : Type) (x : A) (h : M A),
30         catch (pure x) h = pure x;
31 }.
```

Listing 28: The `MonadExcept` typeclass

The only exception that monads supporting the `MonadExcept` effect can throw is `fail`, inherited from the `MonadFail` effect. `MonadExcept` additionally provides an operation `catch` for catching these exceptions. `fail` and `catch` are related by four laws: they form a monoid (`catch_fail_l`, `catch_fail_r` and `catch_assoc`, which intuitively mean that, respectively, failing computations are caught, failing handlers are not used and that the nesting of exception handlers doesn't matter), and they also obey the law `catch_pure` which says that handlers are not used for computations that don't fail. The laws of `MonadFail` are of course inherited too.

Note that the operations supported by `MonadExcept` (`fail` and `catch`) have the same types as those supported by `MonadNondet` (`fail` and `choose`), but their intended meaning is very different, as captured by the laws. It is therefore the operations together with the laws that determine the meaning of an effect, not just the operations alone.

```
Definition work
  {inst' : MonadFail inst} (l : list nat) : M nat :=
    if has 0 l then fail else pure (product l).

Definition fastprod
  {inst' : MonadFail inst} {inst'' : MonadExcept inst'}
    (l : list nat) : M nat := catch (work l) (pure 0).

Theorem fastprod_spec :
  forall
    (inst' : MonadFail inst) (inst'' : MonadExcept inst')
    (l : list nat),
      fastprod l = pure (product l).
Proof.
  unfold fastprod, work; intros.
  case_eq (has 0 l); intros.
    rewrite catch_fail_l, product_has_0.
      reflexivity.
      assumption.
    rewrite catch_pure. reflexivity.
Qed.

Definition next
  {inst' : MonadFail inst} (n : nat) (ml : M nat) : M nat :=
    if beq_nat 0 n then fail else fmap (mult n) ml.

Theorem work_foldr :
  forall (inst' : MonadFail inst),
    work = fold_right next (pure 1).
Proof.
  intros. ext l. induction l as [| h t]; cbn.
    reflexivity.
    unfold work in *. cbn. destruct h as [| h']; cbn.
      reflexivity.
      case_eq (has 0 t); intros.
        rewrite H in *. rewrite <- IHt. rewrite fmap_bind_pure.
          rewrite bind_fail_l. reflexivity.
        rewrite H in *. rewrite <- IHt. rewrite fmap_pure. reflexivity.
Qed.
```

Listing 29: The fast product example

We now proceed to the example, shown in listing 29. Not shown are the functions `product`, which computes the product of a list of natural numbers, `has`, which checks if a natural number is present in a list and the proof of the lemma `product_has_0`, which says that if a zero appears in a list, then the product of that list is equal to zero.

We now define a function `work`, which works for any monad which supports the failure effect. It checks if a list contains a zero and fails if it does and calls `product` otherwise. Using this auxiliary function we define `fastprod`, which wraps `work` in a handler that just returns zero in case of failure.

The correctness of `fastprod` is stated as theorem `fastprod_spec`, which says that `fastprod` is equal to a pure computation that just calls `product`. The proof is very simple. We unfold the definitions of `fastprod` and `work`, introduce variables to context and perform a case split on whether the list contains a zero. In both subcases we use the laws of `MonadExcept` (together with the lemma `product_has_0` in the first case) to conclude that the result is the same as a pure computation using `product`.

This may look unimpressive, because `work` is implemented using a simple case split, without recursion. We could also implement `work` recursively, by checking if the head of the list is zero and failing in this case, otherwise muliplying the product of the tail by the head.

The theorem `work_foldr` states that `work` is equal to a function implemented precisely this way (but using `fold_right` instead of pattern matching). The proof is also very simple: we reason by functional extensionality and then by induction. The empty list case is simple. In the `cons` case we reason by cases on whether the head of the list is zero. In both cases we finish by using some `Monad` and `MonadFail` laws together with the induction hypothesis.

The above proof is as simple as the pen-and-paper one that can be found in [1] in section 5.1 (even though the line of reasoning is a bit different). This shows that at least for simple theorems our library doesn't induce a "formalization overhead".

Chapter 5

Conclusion

We have introduced *hsCoq*, a general purpose Coq library for formally verified programming with Haskell-style abstractions. We have centered our presentation on the matter effects, with which we dealt by implementing a collection of `mtl`-like abstractions whose main inspiration was [1].

In chapter 1 we gave some arguments for why formal verification is worth pursuing and introduced the Coq proof assistant. In chapter 2 we explained the basics of computational effects and outlined the main approaches. In chapter 3 we elucidated the design choices, which weren't really that many, described the implementation and explained the mechanism behind our proof automation. In chapter 4, we have formalized some examples from [1] to show the viability of the library. We also saw some simple theorems which prove our implementation correct, but are also useful from a practical point of view.

5.1 Technical details

The source code and build instructions for *hsCoq* and the source code of this thesis can be found at `https://github.com/Zeimer/HSLib`. The code is extensively commented and this is the only available kind of documentation. Pull requests and suggestions are welcome.

5.2 Related work

There has been relatively little work on effects in Coq. [30] is another attempt at an `mtl`-like library for Coq. It is similar to our work by also drawing inspiration from [1], but differs in technical details (it uses canonical structures instead of classes) and focus (their motivation is formalization of programming languages).

There is an implementation of effects in Coq, available at [27], that allows one to

define programs with IO and even concurrency and then extract the code to OCaml. This development is, however, rather obscure (there is not even a single paper about it). There also is an implementation of exceptions by means of a translation from the Calculus of Constructions to itself, realized in Coq as a plugin [31].

Besides, there has been some work on formally verifying programs with algebraic effects and handlers in Coq [32] [33] and more generally, on effectful programming in Coq [34].

Among dependently typed languages most similar to Coq, there has been some work on an effect system for Idris [35]. Another languages that mixes dependent types and effects (and also refinement types) is F* [36].

5.3 Further work

There's still some work to be done. The TODO list can be found by looking at the `TODO` section in `README.md` and by searching the source code for the keywords `Abort`, `Admitted` and for the marker TODO.

First off, some proofs and instances are missing. For example, I could neither define an instance of `MonadExcept` for `FreeT` nor prove that none exists. The same is true for a few more transformer/class pairs.

Notations could get some love too. The current do notation requires the ugly `;;` for computations that don't bind their result and I couldn't even define the idiom brackets of Idris using Coq's notation mechanism.

Some more theoretical work would be to formally characterize the minimal sets of laws for the various classes and some more practical work would be to implement other useful abstractions known from Haskell, like `Traversable` and `Foldable`.

Another possibility is to rewrite the whole automation engine, so as to use proof by reflection [14], which is more reliable than the more ad hoc tactics written in Ltac. The first step toward this goal has already been made with the tactic `reflect_functor`, which implements basic principles of reasoning with functors.

Last but not least, as already mentioned, *hsCoq* is a great basis on which a library similar to Haskell's `extensible-effects` may be implemented.

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*,
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] Oleg Kiselyov, Amr Sabry, Cameron Swords,
Extensible Effects. An Alternative to Monad Transformers,
http://okmij.org/ftp/Haskell/extensible/exteff.pdf

[21] Oleg Kiselyov, Hiromi Ishii,
Freer Monads, More Extensible Effects,
http://okmij.org/ftp/Haskell/extensible/more.pdf

[22] 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

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

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

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

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

[27] https://coq.io

[28] https://github.com/agda/agda

[29] https://www.idris-lang.org/

[30] Reynald Affeldt, David Nowak,
Experimenting with Monadic EquationalReasoning in Coq
http://jssst.or.jp/files/user/taikai/2018/PPL/ppl2-2.pdf

[31] Pierre-Marie Pédrot, Nicolas Tabareau
Failure is Not an Option: An Exceptional Type Theory,
https://link.springer.com/chapter/10.1007/978-3-319-89884-1_9

[32] Jean-Guillaume Dumas, Dominique Duval, Burak Ekici, Damien Pous
Formal verification in Coq of program properties involving the global state effect,
https://hal.archives-ouvertes.fr/file/index/docid/872324/filename/DDEP-coqstates.pdf

[33] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, Guillaume Hiet,
Modular Verification of Programs with Effects and Effect Handlers in Coq,
https://hal.inria.fr/hal-01799712/document

[34] Greg Morrisett,
Programming with Effects in Coq,
https://link.springer.com/chapter/10.1007%2F978-3-540-70594-9_3

[35] Edwin C. Brady,
Programming and Reasoning with Algebraic Effects and Dependent Types,
https://eb.host.cs.st-andrews.ac.uk/drafts/effects.pdf

[36] https://www.fstar-lang.org/

[37] Matthieu Sozeau, Nicolas Oury, *First-Class Type Classes*,
https://www.irif.fr/~sozeau/research/publications/First-Class_Type_Classes.pdf

[38] https://coq.inria.fr/distrib/current/refman/addendum/type-classes.html

[39] Amokrane Saïbi,
Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories., 1998,
https://tel.archives-ouvertes.fr/tel-00523810/file/Saibi_-_1999_-_Outils_GA_nA_riques_de_modA_lisation_et_de_dA_monstration_pour_la_Formalisation_des_MathA_matiques_en_thA_orie_des_Types_Application_A_la_thA_orie_des_catA_gories.pdf

[40] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, Derek Dreyer,
How to make ad hoc proof automationless ad hoc,
https://software.imdea.org/~aleks/papers/lessadhoc/journal.pdf

[41] Assia Mahboubi, Enrico Tassi,
Canonical Structures for the working Coq user
https://hal.inria.fr/hal-00816703v2/document

[42] https://coq.inria.fr/refman/addendum/canonical-structures.html

[43] Assia Mahboubi and Enrico Tassi with contributions by Yves Bertot and Georges Gonthier,
https://math-comp.github.io/mcb/

[44] Jacek Chrząszcz, *Modules in Coq Are and Will Be Correct*,
https://link.springer.com/chapter/10.1007%2F978-3-540-24849-1_9

[45] https://coq.inria.fr/distrib/current/refman/language/module-system.html

[46] https://wiki.haskell.org/Hask

[47] Miran Lipovača,
Learn You a Haskell for Great Good,
http://learnyouahaskell.com/functors-applicative-functors-and-monoids#applicative-functors

[48] https://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Applicative.html

[49] https://en.wikibooks.org/wiki/Haskell/Applicative_functors

[50] https://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Monad.html

[51] Herman Geuvers,
Induction Is Not Derivable in Second Order Dependent Type Theory,
https://link.springer.com/chapter/10.1007/3-540-45413-6_16

[52] Atze van der Ploeg, Oleg Kiselyov,
Reflection without Remorse,
http://okmij.org/ftp/Haskell/zseq.pdf

[53] https://wiki.haskell.org/ListT_done_right

[54] https://wiki.haskell.org/ListT_done_right_alternative

[55] https://hackage.haskell.org/package/list-t

[56] Georges Gonthier, Assia Mahboubi, Enrico Tassi
A Small Scale Reflection Extension for the Coq system,
`https://hal.inria.fr/inria-00258384v17/document`

[57] `https://coq.inria.fr/refman/proof-engine/ssreflect-proof-language.html`