Program in Coq

Guillaume Claret

USPC
Université Sorbonne
Paris Cité

PARIS DIDEROT

Thèse de doctorat de l'Université Sorbonne Paris Cité
Préparée à l'Université Paris Diderot
École doctorale ED386
IRIF / Inria πr^2

Program in Coq

Par Guillaume Claret

Thèse de doctorat d'informatique

Dirigée par Hugo Herbelin et Yann Régis-Gianas

Présentée et soutenue publiquement à Paris le 18 Septembre 2018

Président	Emmanuel Chailloux	Professeur	Sorbonne Université
Rapporteur	Tristan Crolard	Professeur	Cnam
Rapporteur	Alan Schmitt	Directeur de recherche	Inria Rennes
Examinatrice	Mihaela Sighireanu	Maître de conférence	Université Paris-Diderot
Directeur de thèse	Hugo Herbelin	Directeur de recherche	Université Paris-Diderot
Co-directeur de thèse	Yann Régis-Gianas	Maître de conférence	Université Paris-Diderot

Programmation en Coq

Dans cette thèse, nous cherchons à développer de nouvelles techniques pour écrire plus simplement des programmes formellement vérifiés. Nous procédons en étudiant l'utilisation de Coq en tant que langage de programmation dans différents environnements. Coq étant un langage purement fonctionnel, nous nous concentrons surtout sur la représentation et la spécification d'effets impurs, tel que les exceptions, les références mutables, les entrées-sorties et la concurrence.

Nous travaillons premièrement sur deux projets préliminaires qui nous aident à comprendre les défis existants dans la programmation en Coq. Le premier projet, Cybele, est un plugin Coq pour écrire des preuves par réflexion efficaces avec effets. Nous compilons et nous exécutons les effets impurs en OCaml pour générer une prophétie, une forme de certificat, et interprétons les effets dans Coq en utilisant cette prophétie. Le second projet, le compilateur CoqOfOCaml, importe des programmes OCaml avec effets dans Coq en utilisant un système d'inférence d'effets.

Puis nous décrivons différentes représentations génériques et composables d'effets impurs en Coq. Les calculs avec pause combinent les effets d'exceptions et de références mutables avec un mécanisme de pause. Ce mécanisme de pause permet de rendre explicite les étapes d'évaluation dans le but de représenter l'évaluation concurrente de deux termes. En implémentant le serveur web Pluto en Coq, nous réalisons que les entrées-sorties asynchrones sont l'effet le plus utile : cet effet est présent dans la plupart des programmes et ne peux être encodé de façon purement fonctionnelle. Nous concevons alors les "calculs asynchrones" comme moyen pour représenter et compiler des programmes avec événements en Coq.

Finalement, nous étudions des techniques pour prouver des propriétés à propos de programmes avec effets. Nous commençons avec la vérification du système de blog ChickBlog écrit dans le langage des "calculs interactifs". Ce blog lance un fil d'exécution par client. Nous vérifions notre blog en utilisant une méthode de spécification par cas d'utilisation. Nous adaptons cette technique à la théorie des types en exprimant un cas d'utilisation comme un co-programme bien typé. Grâce à ce formalisme, nous pouvons présenter un cas d'utilisation comme un programme de test symbolique et le déboguer symboliquement, étape par étape, en utilisant le mode interactif de Coq. À notre connaissance, ceci représente la première telle adaptation de la spécification par cas d'utilisation en théorie des types. Nous pensons que la spécification formelle par cas d'utilisation est l'une des clés pour vérifier des programmes avec effets, sachant que la méthode des cas d'utilisation s'est avérée utile dans l'industrie pour exprimer des spécifications informelles. Nous étendons notre formalisme aux programmes concurrents et potentiellement non-terminants, avec le langage des "calculs concurrents". Nous concevons également un vérificateur de modèles pour vérifier l'absence d'interblocage dans un programme concurrent, en compilant la composition parallèle vers l'opérateur de choix non-déterministe.

Mots clés : *Coq, OCaml, programmation fonctionnelle, preuve par réflexion, cas d'utilisation, effets de bord*

Program in Coq

In this thesis, we develop new techniques to conveniently write formally verified programs. To proceed, we study the use of Coq as a programming language in different settings. Coq being a purely functional language, we mainly focus on the representation and on the specification of impure effects, like exceptions, mutable references, inputs-outputs, and concurrency.

First, we work on two preliminary projects helping us to understand the challenges of programming in Coq. The first project, *Cybele*, is a Coq plugin to write efficient proofs by reflection with effects. We compile and execute the impure effects in OCaml to generate a prophecy, a kind of certificate, and then interpret the effects in Coq using the prophecy. The second project, the compiler *CoqOfOCaml*, imports OCaml programs with effects into Coq, using an effect inference system.

Next, we describe different generic and composable representations of impure effects in Coq. The *breakable computations* combine the standard exceptions and mutable references effects, with a pause mechanism to make explicit the evaluation steps in order to represent the concurrent evaluation of two terms. By implementing the Pluto web server in Coq, we realize that the most important effects to program are the asynchronous inputs-outputs. Indeed, these effects are ubiquitous and cannot be encoded in a purely functional manner. Thus, we design the *asynchronous computations* as a first way to represent and compile programs with events and handlers in Coq.

Then, we study techniques to prove properties about programs with effects. We start with the verification of the blog system ChickBlog written in the language of the *interactive computations*. This blog runs one worker with synchronous inputs-outputs per client. We verify our blog using the method of specification by use cases. We adapt this technique to type theory by expressing a use case as a well-typed co-program over the program we verify. Thanks to this formalism, we can present a use case as a symbolic test program and symbolically debug it, step by step, using the interactive proof mode of Coq. To our knowledge, this is the first such adaptation of the use case specifications in type theory. We believe that the formal specification by use cases is one of the keys to verify effectful programs, as the method of use cases proved to be convenient to express (informal) specifications in the software industry. We extend our formalism to concurrent and potentially non-terminating programs with the language of *concurrent computations*. Apart from the use case method, we design a model-checker to verify the deadlock freedom of concurrent computations, by compiling the parallel composition to the non-deterministic choice operator using the language of *blocking computations*.

Key words : *Coq, Ocaml, functional programming, proof by reflection, use cases, side effects*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, we develop new techniques to efficiently write formally verified programs.

1.1 Context

We first introduce some context, be recalling what is a computer program, what kinds of bugs can happen and how formal methods could help us to write safer programs.

1.1.1 Computers and programs

Computers are generic machines designed to mechanically execute programs, which are sequences of instructions such as arithmetical computations, tests and loops. We give the example of the program `sum_squares` (Figure 1.1), computing the sum of the squares of the integers from 0 to n given by the

```
sum_squares(n) :=
  s ← 0
  i ← 0
  while i ≤ n do
    s ← s + i²
    i ← i + 1
  done
  return s
```

Figure 1.1: Program computing the sum of the squares of integers.

$$\begin{array}{rcl} e & := & x \\ & | & \lambda x.\, e \\ & | & e\, e \end{array}$$

Figure 1.2: The shape of a λ-expression.

mathematical formula:

$$\sum_{i=0}^{n} i^2$$

We start by initializing the variable s to 0. The variable s will represent the sum of the squares. Then, we repeatedly add the value i^2 to this sum, for each value of i from 0 to n. In order to repeat this operation, we write a *loop* with the `while` instruction. We execute the operations:

$$\begin{aligned} s &\leftarrow s + i^2 \\ i &\leftarrow i + 1 \end{aligned}$$

while the variable i is lesser or equal to n. Since we initialize the variable i to 0 and increment i by one at each loop step, we effectively add the values i^2 to s for i ranging from 0 to n. Finally, on the last line, we return the value of the variable s holding the sum of the squares. The program sum_squares implements the formula $\sum_{i=0}^{n} i^2$ into a sequence of simple operations runnable by a machine, that is to say an *algorithm*.

Something we can wonder is what kinds of other algorithms can a machine compute? Do we need new instructions to encode more computations? It turns out that all known programming languages are equivalent, in the sense that an algorithm expressed in one programming language can be expressed in any other programming language too[1]. And we need very few instructions in order to encode any algorithms.

For example, an unbounded array of integers, some arithmetic and boolean instructions and a main loop are enough to implement any programs. That is basically how a computer works: the unbounded array is the main memory (the RAM), the arithmetic and boolean instructions are the processor instructions. The main loop keeps the processor active and updates the instruction pointer (the index of the next instruction to execute). Alan Turing formalized this idea with the concept of Turing Machines [60] in 1936, which gives a simple theoretical model of most computers.

The λ-calculus [14], invented in the 1930s by Alonzo Church, is another simple way to represent any algorithms. The λ-calculus is a minimalistic programming language, built entirely on the sole notion of function. In this language, an expression e is either a variable x, a function $\lambda x.\, e'$ which associates, to the variable x, the expression e' or an application $e_1\, e_2$ of the function e_1 to the parameter e_2 (Figure 1.2). We evaluate a λ-expression as we would evaluate a

[1] Of course, programming languages can differ in term of readability, safety or efficiency.

$$\begin{array}{rcl} e_2 & := & ((\lambda f.\, \lambda x.\, f\, x)\, (\lambda x.\, x))\, y \\ & \to & (\lambda x.\, (\lambda x.\, x)\, x)\, y \\ & \to & (\lambda x.\, x)\, y \\ & \to & y \end{array}$$

Figure 1.3: A reduction chain of e_2 in a *call-by-value* strategy.

mathematical function. For example, the expression:

$$e_1 := (\lambda x.\, x)\, y$$

is the identity function applied to y and reduces to y. The second expression:

$$e_2 := ((\lambda f.\, \lambda x.\, f\, x)\, (\lambda x.\, x))\, y$$

is more involved. The expression e_2 is the application of the application of the function $\lambda f.\, \lambda x.\, f\, x$ to the identify function $\lambda x.\, x$ to the variable y. We mechanically reduce e_2 by replacing each sub-expression of the form:

$$(\lambda x.\, e_1)\, e_2$$

by the expression e_1 where x has been substituted by e_2 (Figure 1.3).

The functions are one of the main building blocks of most programming languages, hence we may consider the λ-calculus as the ancestor of the programming languages. Despite being simple, the idea of functions is powerful enough to encode any data-structures. For example, to represent the pair of expressions α and β, we can define:

$$\left\{ \begin{array}{rcl} (\alpha, \beta) & := & \lambda f.\, (f\, \alpha)\, \beta \\ \pi_1 & := & \lambda p.\, p\, (\lambda x.\, \lambda y.\, x) \\ \pi_2 & := & \lambda p.\, p\, (\lambda x.\, \lambda y.\, y) \end{array} \right.$$

These definitions verify the properties:

$$\left\{ \begin{array}{rcl} \pi_1\, (\alpha, \beta) & \to & \alpha \\ \pi_2\, (\alpha, \beta) & \to & \beta \end{array} \right.$$

Using similar encodings[2], we can also represent the booleans, the integers, the lists, etc. For efficiency reasons, actual programming languages also propose

[2] The general idea is to represent a data as its destructor. For example, in OCaml, we would match a pair p with an expression of the form:

```
match p with
| (x, y) ⇒ f x y
end
```

that is to say, when p is (α, β), with:

$$f\, \alpha\, \beta$$

which is the encoding, parametrized by f, of the pair in λ-calculus.

$$\begin{aligned} \Omega &= (\lambda x.\, x\, x)\, \omega \\ &\to \omega\, \omega \\ &= \Omega \\ &\to \dots \end{aligned}$$

Figure 1.4: Infinite reduction chain of Ω.

some built-in and optimized data-structures, even if we could obtain everything from the functions.

A curious expression in the λ-calculus is the following:

$$\omega := \lambda x.\, x\, x$$

The ω expression takes a function x in parameter and applies it to itself. This is counterintuitive because, in math, a function x must have some domain:

$$x : A \to B$$

so x should not be applied to itself since A is not equal to $A \to B$. In the λ-calculus, there are no such restrictions. In particular, we can apply ω to itself:

$$\Omega := \omega\, \omega$$

leading to an infinite reduction chain (Figure 1.4).

We can make use of a type system to prevent us to write curious expressions such as ω. We define a type system by some basic rules associating a type to an expression. The typing rules are mechanically checkable by a computer and ensures that programs are "well-written". By well-written we mean, for example, that the program execution will not get stuck or enter an unexpected infinite loop. The APP rule (Figure 1.5) of the simply typed λ-calculus states that if, in a given context Γ, the expression e_1 has the type of a function from the type A to the type B and the expression e_2 has the type A, then the application of e_1 to e_2 has the type B. Using this rule, we cannot define the ω expression. However, for practical reasons, most programming languages have some unsafe operators to avoid the strictness of the typing rules[3]. One of the biggest challenges in the design of a programming language is to find the right balance between the safety provided by the type system and its simplicity.

So far we have seen how to implement an algorithm and that any programming language can express any algorithm. However, as such, a program is useless as long as it does not communicate with the user. Programming platforms offer various ways to communicate with the outer world. Unlike for the algorithms, two different programming platforms may express different sets of

[3] For example, in OCaml, we can set any type to any value using:

$$\text{Obj.magic} : \forall\, \alpha\, \beta,\ \alpha \to \beta$$

$$\textsc{App}\ \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}$$

Figure 1.5: Application rule of the simply typed λ-calculus.

interactions with the outer world. For example, a JavaScript program in a web page is sealed and should not be able to access to the file system of the client, whereas a Unix shell script must have access to the local file system. In the x86 assembly language[4], the language to which most computer programs are ultimately compiled to, there are three ways to communicate with the outer world: either with the special IN and OUT primitives to emit or receive a byte on a canal, with the interruptions mechanism by which the peripherals can call an assembly function at any moments, or with the DMA[5] system to share the access to a part of the memory with the peripherals. However, many theoretical programming languages such as the λ-calculus do not provide primitives for inputs–outputs operations. One of the main subject of this thesis is the design of an inputs-outputs mechanism for the Coq[6] programming language.

1.1.2 Bugs

Computer programs tend to be complex and can go wrong. A bug occurs when a program does not do what it was supposed to do. In other words, a program has a bug when it does not follow its specification. For example, a program may suddenly halt because it tried to add an integer with a boolean, or may forget to display "Logged in" once a user is authenticated. Many techniques were developed in order to prevent bugs, mainly in the form of better programming languages, practices and testings. According to Dijkstra[7]:

> "Those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with."

Programming languages can forbid, by-construction, whole classes of runtime errors by detecting them or preventing them at compile time. For example, with a sound type system, supposing that the addition is of type:

$$+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

meaning that the addition is parametrized by two integers, the following program would be ruled out during compilation:

$$12 + \texttt{false}$$

[4] The assembly language is the language understood by a microprocessor. The x86 family of processors is mainly used by the PC architecture.
[5] DMA stands for Direct Memory Access.
[6] The Coq language is available on coq.inria.fr under LGPL license.
[7] Said in his Turing award speech.

1. the program displays the login page
2. the user enters a valid couple of login and password
3. the program displays "Logged in"

Figure 1.6: Example of use case for a login system.

because the term `false` is not an integer. In a language without a type system such as Ruby[8], we would get the following runtime error:

`TypeError` : `false` can't be coerced into `Fixnum`

and the program would suddenly halt.

Other kinds of errors are less obvious, and require a human intervention to be classified as a bug or a feature[9]. For example, if a program does not display "Logged in" once a user is authenticated, should it be considered as a bug? In general, we consider that a program behavior is a bug when it does not respect its specification. We can employ many techniques to describe a program specification, like the technique of *use cases* [34] which we study in this thesis. Basically, a use case describes a scenario of expected interactions between a program and its environment. As an example, we give a use case of a login system (Figure 1.6). To verify this use case, we can either run our program by hand on some examples or write a testing program, simulating the behavior of the user. Using the testing program, we can quickly test different couples of login and password and test the program for each new update[10]. However, with this technique, it is impossible to check that the program behaves correctly on each possible instance of login and password, since there are infinitely many.

1.1.3 Formal systems

The *formal systems* are a kind of tool which we use to verify, with a high degree of certainty, that a program respects its specifications in each possible configuration. Formal systems are abstract models of reasoning, based on mathematical methods. In this thesis, we study the formal system Coq.

The system Coq is based on the CiC[11], which is a formalism expressing both programs and mathematical proofs in a unified way, following the Curry-Howard correspondence (Figure 1.7). The idea behind the Curry-Howard correspondence is to realize that the λ-calculus, plus some typing rules, can represent the mathematical logic. For example, for two propositions A and B, a proof of $A \Rightarrow B$

[8] The Ruby language is available on www.ruby-lang.org under Ruby, GPL or BSD license.

[9] "It's not a bug, it's a feature" is a popular joke among programmers.

[10] The technique of rerunning the tests for each update of a program is called continuous integration. The aim is to verify that no new bugs are introduced by a change in the source code.

[11] CiC stands for Calculus of Inductive Constructions.

Program	Logic
$A \to B$	$A \Rightarrow B$
$A \times B$	$A \wedge B$
type	theorem
program	proof

Figure 1.7: The Curry-Howard correspondence.

could be represented by a program which, to a proof of A, associates a proof of B, that is to say a program of type $A \to B$. Similarly, the rule of *modus ponens*, which says that if $A \Rightarrow B$ and A are true, then B is true, can be represented by the function application. Indeed, a function of type $A \to B$ applied to an argument of type A yields a result of type B. Large mathematical theorems were proven using the Coq system, including the theorem of the four colors [26] and the Feit-Thompson theorem.

Formal systems have been used for years to express and verify the specifications of programs. For example, the method B[12] has been used to formally verify the control program of the line 14 of the Parisian metro. However, due to their complexity, the use of formal methods is still limited to critical software. We decide to work of the Coq system since it represents a mature implementation of an expressive, yet conceptually simple, logic with a unified presentation of the notions of proofs and programs. We can use the Coq language in two ways to formally verify programs: either "externally" by modeling a programming language in Coq and by proving properties about this model (deep embedding)[13], or "internally" by using the Coq language as the programming language on which we prove properties (shallow embedding). The "internal" approach is the one we follow in this thesis.

1.2 Effects

One big limitation of the use of Coq as a programming language is that Coq is a purely functional language. Indeed, like in the λ-calculus, there are no inputs–outputs operations in Coq. For example, we cannot write the standard "Hello world" program in pure Coq. Moreover, Coq programs cannot do side-effects like updating a mutable variable, raising an exception or being non-terminating[14]. In particular, all recursive functions must be proven terminating in order to be accepted by the type checker.

These limitations are there for the consistency of the underlying logic. As in math, in Coq a function f is expected to always yield the same result when

[12] An implementation of the method B is available on www.atelierb.eu/en.

[13] See for example the CFML system on www.chargueraud.org/softs/cfml.

[14] The language Coq is stricter than Haskell, in the sense that an Haskell program may raise an exception or not terminate.

applied to some argument v. By contrast, in most programming languages the result may depend on the time, the current user inputs, etc. Moreover, a mathematical function f cannot encounter errors or enter in an infinite loop. This restricts a lot the expressive power of the Coq programming language. The proof language of Coq follows the same limitations, since the proof languages of Coq is the programming language of Coq thanks to the Curry-Howard correspondence. The fact that each proof is evaluated to a unique value, like for a mathematical function, implies the consistency of the logic of Coq. Indeed, if there was a proof p of False:

$$p : \text{False}$$

then we would be able to evaluate p to one of the values of False, which by definition do not exist.

Thus we need to extend the expressiveness of the Coq language while keeping its consistency, in order to make the language usable to write realistic programs. We can make conservative extensions of a pure language through an effect system, which tracks the effects of a program. An effect is any impure action: an input–output operation with the environment, the update of a mutable variable, the launch of an infinite loop, etc. We identify some properties which an effect system may respect. As we study these properties, we also present some existing effect systems.

1.2.1 Visible

The effects given by an effect system may be visible in the type system of the programming language. This is the case of most effect systems we have found in the literature. For example, in the Haskell programming language[15], the printing function is of type:

$$\text{putStr} : \text{String} \to \text{IO}\,()$$

The function putStr takes a string and returns the unit value () doing the IO effect, that is to say interacting with the terminal to display a string. Some programming languages even have special constructs dedicated to the expression of effects. For example, in the PureScript programming language[16], the division function has the type:

$$\text{divide} :: \forall\, e.\ \text{Int} \to \text{Int} \to \text{Eff}\,(\text{err} :: \text{EXCEPTION} \mid e)\,\text{Int}$$

This means that the function divide takes two integers and returns one integer. The function divide makes an exception effect (in case of division by zero) plus any set of effects e, in case the context needs the effects of divide to be lifted to a richer set of effects. The notation:

$$(e_1 :: k_1, \ldots, e_n :: k_n \mid e)$$

[15] The Haskell programming language is available on www.haskell.org.

[16] The PureScript programming language is available on www.purescript.org under MIT license.

stands for a *row*[17] of effects, that is to say a commutative list of named effects plus an unknown list of additional effects e.

1.2.2 Primitive

We may classify the effects of a programming language in three categories: the defined effects, the primitive effects and the declared effects.

The *defined* effects are expressed using other existing programming constructs. In this case, the language implements the effects through some syntactic sugar or through a programming library. A property of the defined effects is that they do not affect the implementation of the underlying programming language. This simplifies the distribution of the effects as a library since their definition does not change the language. This also ensures that the properties of the type system of the language are still valid[18]. However, defined effects cannot extend the expressiveness of a programming language.

The *primitive* effects are defined in the semantics or the implementations of the programming language. This is the case of most practical languages, since many effects cannot be (efficiently) defined in a purely functional way. For example, in Haskell, the IO effect[19] is a primitive of the language. A potential problem of the primitive effects is that they make the semantic of the language more complex, and may break some properties of the type system.

Finally, the *declared* effects are a form in-between the defined effects and the primitive effects. The declared effects are introduced by some abstract primitives whose definitions are left as parameters of the system. These definitions may or may not be expressed later. An advantage of the declared effects is that they do not require to modify the implementation of the programming language while being able to express non-definable effects, such as inputs–outputs in a purely functional language. In a sense, using declared effects instead of primitive effects is like using hypothesis rather than axioms in logic.

1.2.3 Composable and commutative

In some effect systems, we can compose the effects in a commutative way. If we combine two expressions e_1 and e_2 of effects, respectively, ϵ_1 and ϵ_2, the resulting expression should have an effect which includes the effects ϵ_1 and ϵ_2, since we run both the effects of e_1 and e_2. The expression and the meaning of the combination of ϵ_1 and ϵ_2 depends on how we define the effects.

A classical way to represent an effect ϵ is to use a *monad*. A monad is a parametrized type, with $\mathcal{M}\ A$ being the type of the expressions returning a value of type A by doing an effect ϵ. A monad must provide the two operators return and bind (Figure 1.8). The return operator lifts a pure expression to an effectful

[17] In PureScript, the rows are also used to type the objects of the underlying JavaScript.

[18] This is important for us since, in Coq, the correctness of the type system is equivalent to the logical consistency of Coq.

[19] The IO effect stands for inputs–outputs, but is also used to compile any other effects, such as the state with the ST effect.

$$\begin{cases} \text{return} : A \to \mathcal{M}\, A \\ \text{bind} : \mathcal{M}\, A \to (A \to \mathcal{M}\, B) \to \mathcal{M}\, B \end{cases}$$

Figure 1.8: Monadic operators.

$$\begin{cases} \mathcal{S}_S\, A := S \to (A \times S) \\ \text{return}\, x := \lambda s.\, (x, s) \\ \text{bind}\, x\, f := \lambda s. \\ \quad \texttt{let}\ (v, s') := x\, s\ \texttt{in} \\ \quad f\, v\, s' \end{cases}$$

Figure 1.9: The state monad.

expression. The bind operator sequences two effectful expressions, by binding the result of the first one into the second one. The Haskell language popularized the use of a monadic system to represent effects.

As an example, we explain how to compose in two different ways a state effect with an exception effect by using the concept of monads. We represent a state effect of type S, that is to say a mutable reference of type S, by the monad $\mathcal{S}_S$ (Figure 1.9) which takes a state value and returns both the result of an expression and the updated state value. We represent the exception effect with an error of type E by the monad $\mathcal{E}_E$ (Figure 1.10), which either returns a correct value or an error.

A first way to combine the state monad and the error monad is to lift the effectful expressions to the monad $\mathcal{M}_{S \wedge E}$ (Figure 1.11), which represents expressions which may update a state value *and* return an error. This is the way the effects are combined in most programming languages. But another less usual way is possible by lifting effectful expressions to the monad $\mathcal{M}_{S \vee E}$ (Figure 1.12), which represents expressions which may update a state *or* return an error, but cannot do both at the same time. This monad can be useful to represent transactional memories, where an operation either succeeds and may update the state or fails and does not change the state.

The *monad transformers* [41] are a tool designed to compose a specific monad with any other monad. Depending on the order in which we compose the

$$\begin{cases} \mathcal{E}_E\, A := A + E \\ \text{return}\, x := \texttt{left}\, x \\ \text{bind}\, x\, f := \\ \quad \texttt{match}\ x\ \texttt{with} \\ \quad \mid \texttt{left}\, v \Rightarrow f\, v \\ \quad \mid \texttt{right}\, e \Rightarrow \texttt{right}\, e \end{cases}$$

Figure 1.10: The exception monad.

$$\begin{cases} \mathcal{M}_{S \wedge E}\, A := S \to (A + E) \times S \\ \mathrm{return}\, x := \lambda s.\, (\texttt{left}\, x, s) \\ \mathrm{bind}\, x\, f := \lambda s.\, \texttt{match}\, x\, s\, \texttt{with} \\ \quad \mid (\texttt{left}\, v, s) \Rightarrow f\, v\, s \\ \quad \mid (\texttt{right}\, e, s) \Rightarrow (\texttt{right}\, e, s) \end{cases}$$

Figure 1.11: The state *and* exception monad.

$$\begin{cases} \mathcal{M}_{S \vee E}\, A := S \to (A \times S) + E \\ \mathrm{return}\, x := \lambda s.\, \texttt{left}\, (x, s) \\ \mathrm{bind}\, x\, f := \lambda s.\, \texttt{match}\, x\, s\, \texttt{with} \\ \quad \mid \texttt{left}\, (v, s') \Rightarrow f\, v\, s' \\ \quad \mid \texttt{right}\, e \Rightarrow \texttt{right}\, e \end{cases}$$

Figure 1.12: The state *or* exception monad.

monad transformers of the state effect and of the exception effect, we obtain the monad $\mathcal{M}_{S \wedge E}$ or the monad $\mathcal{M}_{S \vee E}$. The monad transformers are interesting because they allow to compose a monad with any other monad. However, the composition of the monad transformers is non-commutative in general, which is both a strength (since we can obtain various effects from more elementary ones) and a weakness (since there are no canonical ways to compose the effects).

The *algebraic effects* [51] aim to overcome the challenges of the composition of monads. The main idea is to separate the declaration from the definition of effects, and to enforce a unified way to define the effects. Indeed, in the framework of the algebraic effects, we define the effects as *handlers* of some operators. An operator is an abstract function parametrized by an argument (Figure 1.13). The handler of an operator depends on three kinds of parameters: the parameter of the operator x, the current continuation k and the current resource r (Figure 1.14). We may call the continuation k zero times (to encode an exception), one time or even several times to encode non-determinism. We use the resource r to represent a mutable state. The expression of a handler can make use of other operators or of effectful primitives of the host language.

As a result, with the algebraic effects, we combine the effects in a commutative way by combining the effectful operators in a program. The effect of a program is the set its effectful operators. Using the handlers mechanism, we compile the effectful operators down to the base language, which may or may not be purely functional.

$$\ldots\, (\texttt{op}\, e)\, \ldots$$

Figure 1.13: Usage of an effectful operator `op` in a source code.

$$\text{handler}_{\mathsf{op}}\, x\, k\, r := \ldots$$

Figure 1.14: Definition of the handler of the operator op.

1.2.4 Inferred

In some effect systems, we can *infer* the effects rather than just state them. As a result, we do not need to write more than we would write in a language without an effect system. There are at least three elements to infer: the resulting effects of a program, the way to compose two effects and the way to sequence subprograms, that is to say the order in which two subprograms are executed.

In the language Haskell, one can infer the effects of a program. Indeed, the programmer represents the effects in the type system using monads, and the type system of Haskell is (mostly) inferable. However, the programmer has to explicit the way he composes and sequences the effects. One composes two different effects using explicit lift operators, to lift the effects of two programs to a common effect. To sequence the effects, there is the explicit do-notation, where each end of line represents the application of the monadic operator bind (which sequences two programs in order).

In the language Koka[20], one infers both the effects and their combinations. The programmer composes the effects in a commutative way as an effect is an unordered set of primitive effects. To infer the sequencing of two effects, one needs to choose a strategy of evaluation. For example, in the expression:

$$(\lambda x.\, e_1)\, e_2$$

one can first substitute the free occurrences of the variable x in e_1 by e_2 (the *call-by-name* strategy), or first evaluate e_2 (the *call-by-value* strategy) or even start by evaluating e_1. In all the languages with inference of the sequencing of effects which we encountered, the strategy of evaluation was call-by-value.

1.2.5 Purifiable

We can hide in a safe way some of the impure effects, so that the resulting program appears pure. We decide to call these effects *purifiable*, since a program containing a hidden effect is, observationally, purely functional. Purifiable effects matter, because we may want to use impure effects in a library for efficiency reasons while providing a purely functional API.

For example, when we catch all the potential exceptions of a program, the resulting program becomes free of exceptions. Hence the exception effect is purifiable. To hide a mutable state in a computation, we can give to this state an initial value and discard the resulting state at the end of the computation. We can often purify an effect by expressing it in a purely functional way (using

[20] An implementation of the programming language Koka is available under Apache license on research.microsoft.com/en-us/projects/koka.

for example a monad), at the expense of some losses of performance. However, we cannot completely purify some effects like the inputs–outputs, since they depend on interactions with the external world.

1.2.6 Certifiable

We use the type systems to specify the kind of values returned by a program. Likewise, we can use the effect systems to specify and verify that a program runs a precise set of effects. We could also ask for more. For example, it may not be enough to know that a program uses a mutable state of type integer. We may also want to know if the program increases the value of this state. More generally, we way want to check any property relating the value of the state before and after the execution of the program.

The project Ynot[21] explores in Coq the use of a state monad accompanied with arbitrary specifications. The user expresses the specifications as Hoare triples[22]. The Ynot library represents the state as a heap of typed values, and relies on the *separation logic* [53] to reason about this heap. An extension of the Ynot monad [42] allows the expression of invariants over the trace of the inputs–outputs operations. In this extension, inputs–outputs operations are axioms specified by their trace.

The project FCSL [55] proposes a Coq framework to represent concurrent programs with a shared heap and synchronization primitives. The reasoning tools of FCSL enable the verification of fine-grainded concurrent and imperative algorithms, represented in the purely functional language Coq with some special primitives.

1.2.7 Compiled

We write programs to eventually run them (hopefully). Programming languages with an effect system may or may not be directly runnable. For example, we can compile purely functional Coq programs to OCaml in order to efficiently execute them [40]. But if we want to add effectful primitives to Coq, we need to explicitly define how to compile these effectful primitives. As long as we do not define how to compile the effectful primitives, we only have a *model* instead of an *implementation* of a program.

When we certify a program with effects, we may also want to certify the compilation chain which we use to run the program. Only a few proven-correct compiling chains are available for programs with effects. The CompCert[23] compiler is a certified compiler for the language C, in the sense that the possible behaviors of the generated assembly code are acceptable behaviors for the source code [39]. A *behavior* contains the impure effects which we can observe from

[21] The results of the project Ynot are available on ynot.cs.harvard.edu.

[22] A Hoare triple is a pre-condition, a computation and a post-condition. We usually express the pre-condition and the post-condition on the program state.

[23] The CompCert compiler is available under a proprietary license on compcert.inria.fr.

a program, that is to say the final status of the program (terminated, non-terminating, terminated with an error) and its trace of inputs–outputs. In this sense, CompCert is a certified compiler for programs with effects.

1.3 Asynchronous inputs–outputs

One of the main kinds of effects which we consider are the asynchronous inputs–outputs. This is because we think that asynchronous inputs–outputs effects are necessary in most programs, while many other effects (like state or exceptions) can be encoded in a purely functional manner. We often need the inputs–outputs to be asynchronous, in order not to block the user interactions (for example in a program with a user interface) or to be able to interact with many other programs (for example in a server connected to multiple clients).

There are at least two approaches to implement asynchronous inputs and outputs:

- with blocking system calls and several system threads;
- with a single system thread and an event loop.

We prefer to follow the second approach because event systems seem to be more efficient and system threads seem hard to reason about. Surprisingly, in term of implementation, not all operating systems propose a convenient event-based API[24]. Fortunately, libraries such as libuv[25] offer an abstraction over Linux, Windows and macOS to provide a unified and optimized event-based API for files and sockets. This library is mainly used by Node.js but bindings are available for other languages.

Here is an example in a pseudo-language of a server logging on the standard output the content of some requested files:

```
while true do
  event := getLastEvent()
  match event with
  | Request fileName ⇒ send (ReadFile fileName)
  | GotFileContent content ⇒ send (LogToStandardOutput content)
  end
done
```

We use an event loop to either start reading a file on a new request, or log the content of a file once it is read. The `Request` events come from some connected clients, and are transferred to our program through the OS. We send commands

[24] See for example this blog post on an attempt to use asynchronous file I/O for libtorrent, which resulted in the choice of blocking I/O with a thread pool: blog.libtorrent.org/2012/10/asynchronous-disk-io/.

[25] The C library libuv is available under MIT license on github.com/libuv/libuv.

to the OS using the `send` primitive. When the OS finishes to read a file, we get a `GotFileContent` event and log to the standard output the content of the file.

This program is asynchronous: the `send` function does not block. The only blocking function is `getLastEvent`, which blocks while there are no event left. The program pauses without using CPU time, and is awaken by the OS once a new event arrives. A difficulty in reading such program is to relate the code sending a command to the OS:

$$\text{send}\,(\texttt{ReadFile}\,\text{fileName})$$

to the code handling the answer:

$$\mid \texttt{GotFileContent}\,\text{content} \Rightarrow \ldots$$

If we take the example of JavaScript, many asynchronous APIs relate the command and the answer through the use of *handlers* (or *callbacks*). To log the content of a file in Node.js we can write:

```
1  fs.readFile(fileName, (error, data) => {
2    if (!error) {
3      console.log(data);
4    }
5  });
```

In the runtime of JavaScript there is still a event loop, but it is invisible to the programmer.

One drawback of the handlers approach is that APIs may not be uniform: some functions expect the handler to be either the first argument or the last one, or require two handlers, one for the successful case and one for the erroneous case. The Promise API[26] proposes a unified representation of asynchronous functions. An asynchronous function returns a *promise*, that is to say a value which may or may not be computed yet. With the primitive .then we get a handler to wait for a promise to be fulfilled and sequence two asynchronous operations:

$$promise.\texttt{then}(result \Rightarrow ...)$$

With Promise.all we can concurrently call a list of promises and return a promise representing the list of results:

$$\texttt{Promise.all}(promise_1, promise_2, ...).\texttt{then}(results \Rightarrow ...)$$

To log a file with a promise we could write:

```
1  fs.readFileAsPromise(fileName).then(data => {
2    console.log(data);
3  });
```

[26] The Promise API was introduced in JavaScript version 6.

Finally, to prevent the "callback hell" or the imbrication of too many handlers:

```
1  asyncCall1(arg1).then(result1 =>
2    asyncCall2(arg2).then(result2 =>
3      asyncCall3(arg3).then(result3 =>
4        asyncCall4(arg4).then(result4 =>
5          ...
6        )
7      )
8    )
9  )
```

the version 8 of JavaScript introduced the async/await notation[27]. With this notation, asynchronous programs are syntactically close to synchronous ones:

```
1  const result1 = await asyncCall1(arg1);
2  const result2 = await asyncCall2(arg2);
3  const result3 = await asyncCall3(arg3);
4  const result4 = await asyncCall4(arg4);
```

The await operator takes a promise as a parameter, waits for it to fulfill and returns its result. We can see the async/await notation as similar to the monadic do notation in Haskell, with the operator .then being the monadic *bind* and the definition:

```
1  async myAsyncFunction(arg) {
2    const result = await asyncCall(arg);
3    ...
4  }
```

being the equivalent of:

```
1  myAsyncFunction arg = do
2    result <- asyncCall arg
3    ...
```

in Haskell.

In the OCaml language, there is the Lwt library [62] to program with promises and asynchronous inputs–outputs. The type of promises returning a value of type α is:

$$\mathrm{Lwt.t}\,\alpha$$

and the notation to sequence two promises:

```
1  let myAsyncFunction arg =
2    let%lwt result = asyncCall arg in
3    ...
```

[27] A similar async/await notation exists in C♯ starting from version 5.

In this thesis, we use the Lwt library to implement some of our examples in OCaml and take some inspirations from its API.

1.4 Main constributions

In this thesis, we explore the design space of the effect systems in Coq. In our opinion, there are no effect systems better than others in absolute terms, because it all depends on the usage. Thus, we guide our research by application cases. For each case, we attempt to implement a safe and usable solution. We summarize here our main contributions.

1.4.1 Formal use cases

Our principal contribution is a formalization of the use cases method in type theory (Chapters page 99 and page 115). As an application, we formally certify the interactive blog system ChickBlog[28] and provide the library Coq.io[29] to write and certify concurrent programs with inputs–outputs in Coq. We briefly present the idea.

In a first approximation, we can consider that the main advantage of formal verification over testing is that we can *cover all the execution paths*. We can only run tests a finite amount of times, and as Dijkstra said:

> "Program testing can be used to show the presence of bugs, but never to show their absence!"

However, a complete paths coverage is not enough to have an exhaustive verification. For example, a sound type system ensures that the types checked at compile time remain valid for any execution paths, but a well-typed program is not necessarily bug-free. Some execution paths may not even be relevant, because of pre-conditions over the system. What probably matters the most is that the formal verification *covers all the intuitive specifications of the program*. By intuitive specifications, we mean what the programmer or the user intuitively expects about the behavior of the program.

To understand what kinds of behaviors we might expect for a program, we can look at the informal specifications written by programmers or clients. There are, for example, the *invariants*, which are properties supposed to hold in each execution context, and the *use cases*, which are scenarios describing the interactions of a program with the external world. Using testing methods, we can check the invariants with code assertions and check the use cases with integration testing. On the side of formal methods, many techniques exist to

[28] The blog system ChickBlog is available under MIT license on github.com/clarus/coq-chick-blog.

[29] The library Coq.io is available under MIT license on coq.io.

1. the user enters a filename f
2. the program reads the file named f
3. the system successfully returns the file content c
4. the program displays the file content c

Figure 1.15: Informal use case for a program cat.

verify invariants, such as the Hoare logic as implemented in Why3[30]. However, we feel that less tooling is available for the formal study of use cases.

As an example, we consider a use case for a program cat displaying the content of a file (Figure 1.15). We cannot, by testing alone, completely verify this use case since there can be infinitely many filenames f and file contents c. To formally verify this use case, we first implement the cat program in Coq using our effect framework Coq.io. Then we implement the use case in Coq as a testing program, keeping the variables f and c symbolic instead of randomly generated. The program of the use case follows a structure in mirror of the cat program: for each system call of the cat program, the use case program generates an answer. The typing rules are such that the use case is verified if we are able to write it. This method extends to recursive and concurrent programs.

In addition, by exploiting the interactive proof mode of Coq, we obtain a symbolic debugger to write the use cases. We symbolically step through each system call of the tested program, and answer to each call by a symbolic response (formalizing the scenario of the use case) in order to go to the next call. If we are able to finish this question–answer game, then the use case is valid. The symbolic debugger also extends to recursive and concurrent programs.

The Coq.io library is not limited to the verification of use cases. First of all, we can use Coq.io to write unverified concurrent programs with inputs–outputs directly in Coq, gaining the fact that, by construction, Coq programs terminate and cannot raise uncaught exceptions. Secondly, since we express the Coq.io programs in Coq, we can continue to use standard Coq techniques to verify the purely functional parts of the programs. Finally, we can also express and verify temporal invariants over Coq.io programs since their structure is explicit.

1.4.2 Cybele

In a joint work with *Yann Régis-Gianas*, *Lourdes del Carmen González Huesca* and *Beta Ziliani*, we designed the Cybele[31] plugin for Coq, which implements a new technique to do proofs in Coq with the use of *simulable monads* (Chapter page 27). In Cybele, we can write proofs by reflection (proofs using a certified decision procedure) including impure effects, such as mutable references or calls

[30] The Why3 platform is available under LGPL license on why3.lri.fr.
[31] The Coq plugin Cybele is available under MIT license on cybele.gforge.inria.fr.

to an unverified oracle, in order to improve the verification time. We first execute the proofs in OCaml for maximum speed, and then check the proofs in Coq by reusing some intermediate results from the OCaml execution.

1.4.3 Compile OCaml to Coq

We propose a compiler[32] importing OCaml programs into the language Coq. This compiler infers the effects of an OCaml program (exceptions, global references, non-termination and inputs–outputs) and generates a corresponding Coq program with an explicit effect system (Chapter page 47). We do not handle all of the OCaml language. In particular we do not handle the functors.

1.4.4 List of publications

- Guillaume Claret, Lourdes Del Carmen González-Huesca, Yann Régis-Gianas, and Beta Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. In *ITP*, volume 7998 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2013. doi:10.1007/978-3-642-39634-2

- Guillaume Claret and Yann Régis-Gianas. Mechanical verification of interactive programs specified by use cases. In Stefania Gnesi and Nico Plat, editors, *3rd IEEE/ACM FME Workshop on Formal Methods in Software Engineering, FormaliSE 2015, Florence, Italy, May 18, 2015*, pages 61–67. IEEE Computer Society, 2015. doi:10.1109/FormaliSE.2015.17

1.5 Reading guide

We provide here some reading guide so that the interested reader can skim to the parts which are the most relevant to him.

1.5.1 Chapters

We give a brief overview of the content of each chapter, which tools and techniques we use and what they achieve.

Cybele (chapter 2) We design a tool Cybele to help to make proofs by reflection in Coq. Proofs by reflection are proofs made by a decision procedure, written and proven correct in Coq. The tool Cybele, along with the concept of *simulable monad*, allows to write decision procedure with side-effects like mutations or non-determinism. These side-effects help to build efficient decision procedures in the purely functional language Coq. We first compile and execute

[32] The compiler coq-of-ocaml is available under MIT license on github.com/clarus/coq-of-ocaml.

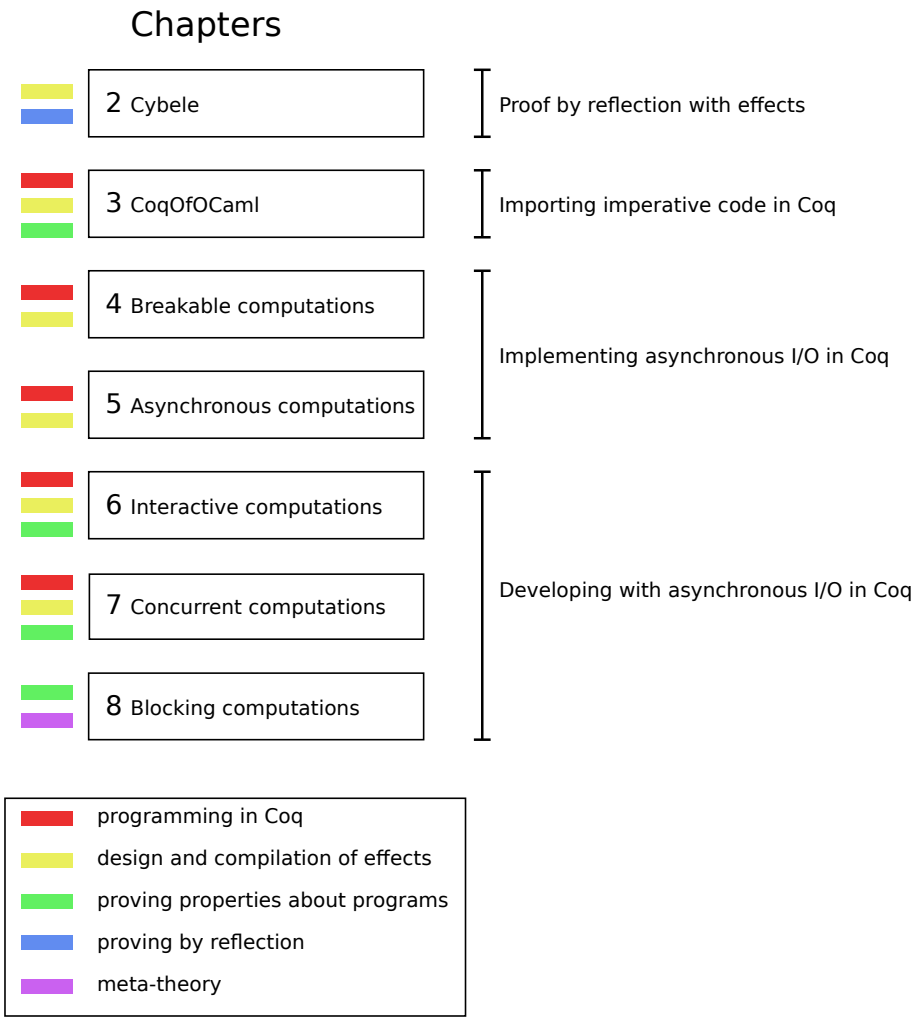

Figure 1.16: Reading guide.

these procedures in OCaml to generate a *prophecy*, and then a second time in Coq to conclude the proof, using the prophecy.

CoqOfOCaml (chapter 3) We present the compiler CoqOfOCaml to compile programs written in a subset of OCaml to Coq. We describe the compilation chain which includes a basic effect inference mechanism. The target language is Coq with side-effects encoded in a purely functional and composable way. We present this encoding and how the compiler composes the effects. The aim of CoqOfOCaml is to import existing OCaml code to Coq to then prove some properties on it. Once we imported the code to Coq, we can modify it and compile it back to OCaml thanks to the extraction mechanism. Thus CoqOfOCaml may also be useful for Coq programers who want to reuse existing algorithms written in OCaml. One major limitation of CoqOfOCaml is that it does not compile the OCaml functors.

Breakable computations (chapter 4) The *breakable computations* are a first attempt to define a generic notion of programs with side-effects written in Coq. These computations can represent composable mutations and exceptions with the ability to pause the execution at any time. We use this pause mechanism to implement a scheduler in Coq.

Asynchronous computations (chapter 5) The *asynchronous computations* are a way to represent programs with asynchronous inputs–outputs in Coq. The main primitive is the call of a system function together with a handler to listen to the results. We decided to focus on the effect of asynchronous inputs–outputs and implemented a Web server in the language of asynchronous computations. We propose a compilation chain to OCaml to execute the programs with effects. We implement the handler mechanism in Coq supposing the existence of an event loop. We compile the event loop to OCaml using the extraction mechanism of Coq.

Interactive computations (chapter 6) The *interactive computations* are a representation in Coq of programs with sequential inputs–outputs. We introduce with this representation a technique of verification by *use cases*. We verify programs specified by sequences of interactions with the external world. Then, we compile interactive computations to OCaml with the extraction mechanism and rely on the OCaml library Lwt to implement the inputs–outputs. We give the example of a small blog system written and verified in Coq.

Concurrent computations (chapter 7) The *concurrent computations* are an extension of the interactive computations to add a concurrency operator and have asynchronous inputs–outputs. We show that the technique of verification by *use cases* extends naturally to the concurrent computations. We present different patterns to compose effects and architecture programs. We compile down Coq programs with effects to OCaml using the Lwt library to implement

concurrency and inputs–outputs. This is the final chapter in term of implementation of a framework to program with effects in Coq. The result is available as the Coq.io library on http://coq.io/.

Blocking computations (chapter 8) The *blocking computations* are similar to the concurrent computations with an added semantics to define blocking interactions. An example of (potentially) blocking interaction is the acquisition of a lock. We compile the blocking computations to a simpler language *choose*, which we prove equivalent. Using this choose language, we implement an example of model checker to check if a program is deadlock-free.

1.5.2 Chapter dependencies

We present the dependencies between the chapters according to the interests of one reader.

Programming in Coq The main chapter about programming in Coq is the chapter 7 introducing the *concurrent computations*. This chapter presents a free monad with asynchronous inputs–outputs to write effective programs in Coq using the library Coq.io[33]. We show how to verify properties expressed as use cases and how to compile Coq programs with effects to OCaml in order to execute them. We can see the chapter 6 about interactive computations as an introduction to the concurrent computations in the special case of sequential inputs–outputs.

The chapters 4 and 5 show other definitions of computations with effects also aimed at programming in Coq. In particular, we study the mutation, the exception, the concurrency and the inputs–outputs effect and propose an implementation to run concurrent inputs–outputs in chapter 5.

In chapter 3 we present a compiler to import existing effectful algorithms written in OCaml to Coq. This compiler may be useful for the user who wants to program in Coq and reuse existing OCaml code.

Design and compilation of effects In most of the chapters we present an effect system to write effectful programs in Coq and a way to execute them. In chapter 2, we show a *simulable* monad with exceptions, mutations, non-termination, and non-determinism. We implement a special execution mode by compilation to OCaml to efficiently execute effectful decision procedures, while preserving the correctness of Coq proofs using these decision procedures. In chapter 3, we design an effect system with non-termination, composable exceptions and mutations to import OCaml programs in Coq with a simple effect inference system.

In chapter 4 we study an effect system with the ability to represent a pause in the execution of an expression. Thanks to this representation of pauses, we implement a concurrent scheduler in Coq. In chapter 5, we study an effect

[33] The library Coq.io is available under MIT license on coq.io.

system with asynchronous inputs–outputs in order to implement a small executable HTTP server in Coq. We representation asynchronous operations with handlers interpreted using an event-loop.

In chapter 6, we continue to study the effect of inputs–outputs in the special case of sequential interactions, with the focus on the development of proofs techniques about specifications of programs with effects. In chapter 7, we extend the previous specification techniques to concurrent and non-terminating programs.

Proving properties about programs We study the proof of properties about programs with inputs–outputs in chapters 6 and 7 with the method of specifications by use cases. We define a use case by a set of possible interactions between a program and its environment. We show a method to validate them using the proof mode of Coq. This is the main contribution of this thesis.

In chapter 3, we propose a tool to import existing OCaml programs to a shallow embedding in Coq. Once imported, we can prove properties about these programs using standard Coq techniques, and compile back these programs to OCaml using the extraction mechanism of Coq.

Proving by reflection In chapter 2, we present the Coq plugin Cybele[34] to make proofs by reflection implementing the concept of *simulable monads*. Using Cybele, a user can write decision procedures with imperative traits like mutations, non-determinism (resolved efficiently in OCaml), and dynamic checks to help the formal verification of decision procedures at the expense of completeness.

Meta-theory For the reader interested by the meta-theory about programs, we study in chapter 8 the trace semantics of two concurrent languages. We show that we can compile the first into the second while preserving the semantics. We use this compilation scheme to design and verify a small model checker for deadlock-freedom.

[34] The project Cybele is available under MIT license on cybele.gforge.inria.fr.

Part I

Proof by reflection with effects

Chapter 2

Cybele

This chapter is a joint work with Yann Régis-Gianas, Lourdes del Carmen González Huesca *and* Beta Ziliani, *published in* ITP 2013. *It is based on an original idea of* Yann Régis-Gianas. *I mainly worked on the implementation and on the experiments with the* Coq *plugin. The results are available on the website of the project on http://cybele.gforge.inria.fr/.*

2.1 Abstract

The proof-by-reflection is a well-established technique that employs decision procedures to reduce the size of proof-terms. We can write decision procedures either in Type Theory—in a purely functional way that also ensures termination— or in an effectful programming language, where they are used as oracles for the certified checker. The first option offers strong correctness guarantees, while the second one permits more efficient implementations.

We propose a novel technique for proof-by-reflection that marries, in Type Theory, an effectful language with (partial) proofs of correctness. The key to our approach is to use our notion of *simulable* monads. We encode several examples using simulable monads and demonstrate the advantages of the technique over previous approaches.

2.2 Introduction

In Type Theory, types may embed computations, thereby allowing for a proof technique called *proof by reflection*. This technique reduces the time to type-check a proof by replacing potentially large proof-terms by small proof-terms, whose verification consists of computing at the type level.

To illustrate the proof by reflection technique, let us say that verifying a proof Δ of $P\,a$ is a computationally expensive task, with:

$$\begin{cases} P & : & A \to \text{Prop} \\ A & : & \text{Type} \\ a & : & A \end{cases}$$

Let B be a type such that there exists an interpretation function I from B to A, and a decision procedure $D : B \to \text{bool}$. Furthermore, let us assume that D decides P, that is to say, there is a theorem:

$$\text{sound} \;:\; \forall\,(x : B),\; D\,x = \text{true} \to P\,(I\,x)$$

which states that for every element x of B, if the decision procedure returns `true` for this element, then the property P holds for the interpretation of x. Then if we have some $b : B$ such that $I\,b = a$, we can replace the original proof-term Δ with:

$$\text{sound}\,b\,(\text{refl\_equal true})$$

where `refl_equal` has type $\forall\,(x : \text{bool}),\; x = x$. Typechecking that the proof-term above has the expected type $P\,a$ effectively amounts to execute the procedure D on b, checking that its result is equal to `true` and checking that the interpretation of b is equal to a[1].

Previous works [30, 5] have exposed several advantages and weaknesses of proof by reflection, especially in comparison with the traditional LCF proof style [28]. In a nutshell, the former is considered more robust to change, while the latter is easier to write. Indeed, proving by reflection has a price. Usually, we must write the decision procedure D in a purely functional programming language, with only total functions and no imperative features. Furthermore, soundness proofs are often complex and thus difficult to construct [27]. These two problems, the restricted language and the need for keeping the proof of soundness simple, incite the proof developer to write inefficient decision procedures, which is regrettable since proof search is intrinsically a computationally expansive process.

There is a variation of proof by reflection that alleviates some of these problems, called certifying proof by reflection [3, 29]. In this technique, we write the decision procedure in a general purpose programming language. The decision procedure acts as an *oracle* and returns a certificate, which we mechanically verify with a proof assistant using a certificate checker written in Type Theory. This checker and its correctness proof are usually kept simple, whereas the untrusted decision procedure can be as sophisticated as necessary in order to be efficient. However, this technique has its drawbacks. First, it is not as efficient as one may expect, as the certificate embedded in the resulting proof-term can be large. In addition, there is a cost in executing the oracle and verifying the certificate with the checker. Second, we force the proof developer to write both

[1] Usually there is also a previous step where a b is constructed for the given a. This step is called reification in the literature.

the certificate checker and the decision procedure (or adapt an existing one in order to produce the certificate). Third, the implementation of an oracle usually gives only weak guarantees about its applicability (a perfectly valid but useless oracle could fail on every input). Indeed, proving completeness properties about a program written in a general purpose programming language is notoriously hard.

In this paper, we propose a novel style of proof by reflection that allows for writing efficient decision procedures in Type Theory. Our idea is to use an (untrusted) compiled version of a monadic decision procedure written in Type Theory as an efficient oracle for itself. Like in the certifying style, we develop the decision procedure within an effectful language. However, unlike in the certifying style, we write the decision procedure in Type Theory, in a language extended with monads as commonly found in Haskell programs [63]. In this way, programmers have a full set of effects at their hand (references, exceptions, non-termination), together with dependent types to enforce (partial) correctness. We automatically compile this decision procedure into an impure programming language with an efficient computational model. We execute the compiled code, and extract a small piece of information (the *prophecy*) to efficiently simulate this execution in Type Theory using the initial monadic decision procedure.

To formalize this idea we define the concept of a posteriori simulation of effectful computations in Type Theory. Roughly speaking, it involves determining, for a computation C encapsulated in the monadic type $M\,A$, the conditions for which there exists a piece of information p such that the evaluation of C, using the prophecy p, can witness an inhabitant of type A.

We believe this technique to be more lightweight than existing approaches. Indeed, we do not require a full proof of correctness or a certificate checker to execute the decision procedure once written in our monad. To sum things up, our contributions are:

- a technique to perform a posteriori simulations of effectful computations in Type Theory in order to promote these computations as genuine proofs by reflection (page 34);
- the plugin Cybele[2] for the Coq proof assistant, which enables the effectful computation as an interactive decision procedure of a Coq function written in monadic style (page 36);
- several examples of proofs by reflection in this new style, showing its simplicity and efficiency (page 39).

2.3 General idea

In this section we give an introduction to our simulation-based style of proof by reflection, in Coq. As a running example, we consider the problem of

[2]The plugin Cybele is available under MIT license on cybele.gforge.inria.fr.

```
decide (⋀_{i∈I} A_i ≤ B_i ⇒ A ≤ B) :=
  let rec traverse : T → bool := λC.
    if C = B then ⊤
    else if marked C then ⊥
    else
      mark C;
      choice D s.t. ∃ j, C ≤ D ≡ A_j ≤ B_j ∧ traverse D in
  traverse A
```

Figure 2.1: Pseudo-code of the procedure `decide`.

determining if a conjunction of inequalities:

$$\bigwedge_{i \in I} A_i \leq B_i$$

between ground terms of type T logically implies $A \leq B$ by transitivity, for some values A and B. A simple decision procedure for this problem boils down to a depth-first traversal of the graph induced by the hypotheses. In the procedure `decide` implemented in pseudo-code, we avoid infinite loops by marking all of the visited terms (Figure 2.1). We cannot directly implement this procedure in Coq since it uses side effects (marks) and is not obviously terminating (of course, it is, but the argument is not syntactical as Coq requires). As mentioned in the introduction, we are going to implement procedures with side effects using a monad $M\,\Sigma\,T'$, where Σ represents the type of the state and T' the returning type of the monad. Then we use this procedure as an oracle for itself, as we are going to see in the second part of this section.

Using our system, we implement the decide procedure in Coq (Figure 2.2). At high level, the code looks like an ML implementation of the pseudo-algorithm, annotated with dependent types. We are going to explain line by line why this procedure is a faithful representation of the pseudo-code shown above, while introducing the notations used in the rest of the paper.

We start by describing the type formula in line 1. It is a record containing a list of pairs of elements (A_i, B_i) – the hypotheses – and a pair of elements (A, B) – the goal. When an element f of this type is interpreted using the function `interpret`, it produces the type $\bigwedge_{i \in I} A_i \leq B_i \to A \leq B$. This is the type returned by the monad.

Line 2 is straightforward: it binds the pair of elements being compared in the goal of f to variables a and b. In line 3, the keyword `letrec!` introduces a (potentially nonterminating) recursive function. Behind this syntactic sugar is hidden the application of a dependently-typed general fixpoint operator. The returning type of the local fixpoint `traverse` is specified between brackets. It returns a proof that the inequality $x \leq b$ holds under the hypotheses of formula f. As we can see in line 13, the argument x is instantiated with the element a from the goal, therefore effectively proving $a \leq b$.

```
1  Program Definition decide (f : formula) : M Σ (interpret f) :=
2    let (a, b) := goal f in
3    letrec! traverse x [ interpret_hypotheses f → x ≤ b) ] :=
4      if x =?= b then return (▷ eq_refl x)
5      else if! marked x then error "Not Found"
6      else do! mark x in
7        choice (interpret_hypotheses f → x ≤ b)
8          (successors x (hypotheses f))
9          (λ(s : {y : T & interpret_hypotheses f → x ≤ y}) ⇒
10           let! Hyb := traverse (π1 s) in
11           return (▷ (λ(hs : interpret_hypotheses f) ⇒
12             le_trans x (π1 s) b)))
13   in ▷ (traverse a).
```

Figure 2.2: Implementation of the `decide` procedure, with ▷ an implicit coercion operator.

```
1  let mark (x : T) : M Σ unit :=
2    let! marks : Marks := !marks_ref in
3    marks_ref :=! MarksTable.add x true marks
4
5  let marked (x : T) : M Σ bool :=
6    let! marks : Marks := !marks_ref in
7    ret (MarksTable.mem x marks)
```

Figure 2.3: The `mark` and `marked` functions.

In line 4 the current element is compared with b, assuming that the type of the elements, T, has decidable equality. If it is equal, then the reflexivity proof is returned using the standard unit monadic combinator return [63]. We defer the explanation of the operator ▷.

In line 5, an error is raised if the element x is already marked. We show the implementations of `mark` and `is_marked` on the Figure 2.3. We use reference reads with the operator ! and reference writes with the operator :=!. In line 6 we mark the element, and in lines 7-12 we try to find a proof of $x \leq b$ by transitivity, by finding a c such that $x \leq c$ and $c \leq b$. For that, we make a list with all the successors of x, that is, all c such that $x \leq c$ is in the list of hypotheses. Then, we call the function `choice` (Figure 2.4). The choice operator iterates over a list to find an element c that successfully produces a result using the function `pred`. At each step of the iteration, the function makes use of the exception mechanism to catch failed attempts and recurse on the tail of the list. Coming back to traverse, in line 10 we call the function recursively using the standard monadic bind operator:

$$\texttt{let!}\ x := e_1\ \texttt{in}\ e_2$$

```
1  Fixpoint choice A {T} (cs : list T) (pred : T → M Σ A)
2    : M Σ A :=
3    match cs with
4    | nil ⇒ Error "Not found"
5    | c :: cs ⇒ try! pred c with _ ⇒ choice A cs pred
6    end.
```

Figure 2.4: The `choice` function.

The resulting proof `Hyb` of $c \leq b$ is then used to prove $x \leq b$ by transitivity. Finally, notice that in line 1 we use the standard Coq keyword `Program` [58]. This keyword allows for writing a partial term, where the holes are exposed to the user as proof obligations. In our case, the holes come from type coercions, noted as $\triangleright$, and they are solved automatically by Coq.

The compiled decision procedure as an oracle The type system of Coq will not let us apply `decide` as it is on some formula f to prove the goal. The reason is simple: an infinite loop would break the soundness of the prover. We need some extra information, which we call *prophecy*, to evaluate `decide`. For instance, in our example this prophecy is the number of steps that leads to a successful result.

To get this information, we execute a compiled version:

$$\mathcal{C}\,\text{decide}$$

in OCaml, which performs the effectful computations. A central property of the system is that $\mathcal{C}$ maps the effectful computations of the monad in Coq to effectful terms in OCaml, in such a way that a relation of a posteriori simulation stands between the compiled term $\mathcal{C}(t)$ and the initial monadic term t. Intuitively, if a compiled term $\mathcal{C}(t)$, with t of type $M\,T$[3] converges to a value v, then the same evaluation can be simulated a posteriori in Coq, using some prophecy p. This prophecy completes the computation t in order to get a term convertible to:

$$\texttt{return}\,t'$$

for some term t' of type T. We instrument the compiled code $\mathcal{C}(t)$ to produce the prophecy along its execution.

Coming back to our example, we present the extracted OCaml code of the function `decide` (Figure 2.5, slightly beautified). The compiled program has almost the same shape as the source term except that every term in `Prop` has been erased and that the primitives of the monad are replaced with combinators defined in OCaml. These combinators implement an effect and contribute to determine the prophecy.

[3] For presentation purposes, we leave out the parameter Σ representing the type of the state.

```
1  let rec fix f x = incr nbstep (); f (fix f) x
2
3  let rec choice cs pred0 = match cs with
4    | Nil → failwith "Not found"
5    | Cons (c, cs0) → try pred0 c with _ → choice cs0 pred0
6
7  let decide f =
8    let (a, b) = goal f in
9    let traverse = fix (fun traverse x →
10     match O.eq_dec x b with
11     | Left → ()
12     | Right → if marked x then failwith "Not found" else (
13       mark x;
14       choice (successors x (hypothesis f))
15         (fun s0 → traverse (π1 s0))))
16 in traverse a
```

Figure 2.5: Extracted code of the function `decide`.

For instance, the `fix` combinator not only implements a general fixpoint but also stores the number of iterations that are performed by the decision procedure in a global variable. Once applied to a specific formula, this compiled function may diverge or fail. In the setting of interactive theorem proving, divergence is not an important issue because we stay in front of the screen waiting for an answer, and we can always interrupt the oracle if it takes too much time to respond. In the case of a successful execution of the decision procedure, a prophecy of type `nat` is extracted from the final value of the mutable cell incremented by `fix`.

The final proof-term The resulting proof-term corresponding to the application of the procedure to some formula f is:

$$\text{unit\_witness}\,(\text{decide}\,f)\,p\,(\text{refl\_equal}\,\texttt{true})$$

where p has type `Prophecy` (in this case, a natural number), and for any type T:

$$\begin{array}{rcl} \text{unit\_witness} & : & \forall\,(x : M\,T),\ \text{Prophecy} \to \text{is\_unit}\,x = \texttt{true} \to T \\ \text{is\_unit} & : & M\,T \to \text{bool} \end{array}$$

The execution time of checking that this term has type:

$$\text{interpret}\,f$$

is split between the execution time of typechecking the prophecy p and the weak head normalization of the procedure, using p to guide the reduction. The overall execution time of the proof-by-reflection results from executing the decision

procedure in OCaml plus typechecking the final proof-term, which as we just mentioned, essentially consists of executing the decision procedure a second time in Coq[4]. One can wonder if it is not a waste of time to execute the decision procedure twice, but, as it turns out, using the hints in p, the execution time of the simulation can be tremendously reduced in comparison with the execution of the oracle (page 41). Putting all the pieces together, our plugin performs the three following steps when proving a goal with a monadic procedure `proc`: translates and compiles `proc` into OCaml; executes the compiled code $\mathcal{C}(\text{proc})$ and obtains prophecy p; builds the proof term:

$$\text{unit\_witness}\,\text{proc}\,p\,(\text{refl\_equal}\,\texttt{true})$$

2.4 Formalization

In this section we summarize the principle of a posteriori simulation of effectful computations. The interested reader is invited to read the full formalization in the companion technical report [16]. We focus on the simply typed λ-calculus, but the presented results are easily extensible to full Coq and OCaml.

We define two languages:

- λ, a purely functional and strongly normalizing programming language with monadic constructs;
- $\lambda_{v,\bot}$, a non-terminating functional programming language.

The definition of λ is parameterized by a monad M, which is abstractly specified by a set of requirements. Accordingly, $\lambda_{v,\bot}$ offers impure operators that match the effectful primitives of the monad M.

The constant $\Downarrow$ of the language λ performs the reduction of a term t using a prophecy p of type P, which we note $\Downarrow_p\ t$. We require the existence of a total order $\leq$ over values of type P and a minimal element $\bot$ for this order. A reduced computation is still a computation, so $\Downarrow$ has type:

$$P \to M\,T \to M\,T$$

We are interested in reasoning on $\beta\delta$-convertibility between terms (where the δ-reduction is the unfolding of constant definitions). We write $\star t$ for $\Downarrow_\bot \text{unit}\,t$.

Definition 1 (Simulable monad)**.** *A type constructor M is a simulable monad if it is equipped with `return`, `bind`, $\Downarrow$ and an associated type for prophecies P, such that the following four requirements are fulfilled.*

Requirement 1 (Standard monadic laws)**.** *M is a monad:*

$$\begin{cases} \text{bind}\,(\text{return}\,t)\,f & = & f\,t \\ \text{bind}\,t\,(\lambda x.\,\text{return}\,x) & = & t \\ \text{bind}\,(\text{bind}\,t_1\,t_2)\,t_3 & = & \text{bind}\,t_1\,(\lambda x.\,\text{bind}\,(t_2\,x)\,t_3) \end{cases}$$

[4] We assume the compilation time not to be significant.

Requirement 2 (Reduction)**.**

$$\begin{cases} \forall\, t, p_1, p_2,\ \Downarrow_{p_1} \mathrm{return}\, t = \Downarrow_{p_2} \mathrm{return}\, t \\ \forall\, t, u, p_1, p_2,\ (p_1 \leq p_2 \wedge \Downarrow_{p_1} t = \star u) \Rightarrow \Downarrow_{p_2} t = \star u \\ \forall\, p,\ \Downarrow_p \mathrm{bind}\, t_1\, t_2 = \Downarrow_p \mathrm{bind}\, (\Downarrow_p t_1)\, t_2 \end{cases}$$

The impure functional and non-terminating language $\lambda_{v,\perp}$ has the same syntax as λ, except that we replace the monadic constants by effectful operators. We equip the language with an instrumented big-step operational semantics for a weak call-by-value reduction strategy. The reduction judgment in this instrumented semantics is:

$$\eta \vdash u \Downarrow_{p \to p'} v$$

which is intended to be read as "the execution of a term u under the environment η converges to a value v and computes a prophecy p' from an initial prophecy p". The purpose of the instrumentation of the compiled code is to monotonically refine the prophecy at each step of the computation:

Requirement 3 (Monotonicity of prophecy computation)**.**

$$\forall\, p, p_1,\ \eta \vdash u \Downarrow_{p \to p'} v \implies p \leq p'$$

We define the compilation function $\mathcal{C}(\cdot)$ from λ to $\lambda_{v,\perp}$:

$$\begin{array}{rclcrcl} \mathcal{C}(x) & = & x & \qquad & \mathcal{C}(\mathrm{return}) & = & \lambda x.\, x \\ \mathcal{C}(\lambda x.\, t) & = & \lambda x.\, \mathcal{C}(t) & & \mathcal{C}(\mathrm{bind}) & = & \lambda x, y.\, y\, x \\ \mathcal{C}(t_1\, t_2) & = & \mathcal{C}(t_1)\, \mathcal{C}(t_2) & & \mathcal{C}(\Downarrow_p) & = & \text{undefined} \end{array}$$

$$\begin{array}{rcl} \mathcal{C}(M\, T) & = & \mathcal{C}(T) \\ \mathcal{C}(C\, \vec{T}) & = & C(\mathcal{C}(\vec{T})) \\ \mathcal{C}(T_1 \to T_2) & = & \mathcal{C}(T_1) \to \mathcal{C}(T_2) \end{array}$$

The translation replaces the monadic constructs `return` and `bind` with their respective definitions in the identity monad, and converts each effectful primitive of the monad to the corresponding impure construction of $\lambda_{v,\perp}$. The type for prophecies is kept fully abstract to the programmer. As a consequence, only the instrumented compiled code is allowed to generate prophecies. Therefore, the compilation of $\Downarrow_p$ is explicitly undefined because this operator cannot appear in a well-typed user-written monadic term. Finally, the compilation of an effectful monadic constant must extend the prophecy in a sufficient way to make the simulation converge.

Requirement 4 (Adequate instrumented compilation)**.**

$$\forall\, p_0, \ldots, p_{n+1}, p, \left.\begin{array}{l} \forall\, i,\ \eta \vdash \mathcal{C}(t_i) \Downarrow_{p_i \to p_{i+1}} v_i \\ \qquad \eta \vdash \mathcal{C}(c(t_0, \ldots, t_n)) \Downarrow_{p_0 \to p} v \end{array}\right\} \Rightarrow\ \exists\, u,\ \Downarrow_p c(t_0, \ldots, t_n) = \star u$$

2.4.1 A posteriori simulation

The main theorem states that, if the evaluation of $\mathcal{C}(t)$ converges for some computation t, then there exists a prophecy p to simulate t back in λ.

Theorem 1 (A posteriori simulation)**.** *Let $\cdot \vdash t : M\,T$ be a computation which compilation converges to a value, that is $\cdot \vdash \mathcal{C}(t) \Downarrow_{p \to p'} v$ holds. Then there exists a term t' such that $\Downarrow_{p'} t = \star t'$.*

We give the proof of this theorem in the companion technical report [16].

2.5 Implementation

We provide a plugin[5] for Coq to develop proofs using the method described in this work. The plugin includes:

- a library with the definition of a simulable monad to write effectful decision procedures;
- a tactic called `cybele` waiting for a monadic term t of type $M\,T$ to try to solve a goal T.

Behind the scene, the tactic compiles the monadic term into an OCaml program, executes this program and if its execution converged, uses the resulting prophecy to produce a proof-term in Coq.

The formal notion of simulable monad served as a guideline for the implementation: we defined a compilation function from Coq to OCaml as well as a simulable monad in Coq that respect the requirements drawn by our formal study. However, to improve the usability and the efficiency of our tool, some practical aspects of the implementation differ from the formal specification.

2.6 A simulable monad in Coq

Our simulable monad combines a partiality monad, with a non-termination monad, a state monad, and a printing monad[6]. The proof that these monads are simulable is given in the companion report of the paper. We parametrize the monad with a signature Σ to type the memory. The type definition of the monad is:

$$M\,\Sigma\,\alpha = \mathrm{State.t}\,\Sigma \to (\alpha + \mathrm{string}) \times \mathrm{State.t}\,\Sigma$$

The monad takes a state and returns a new state plus a value of type α if the computation is successful, or an error message in case of failure. We implement the state as a dependent record containing:

[5] The plugin Cybele is available under MIT license on cybele.gforge.inria.fr.

[6] In this work, we are not interested in a fine grain control of effects so we provide only one monad with all the effectful operations we found useful.

- the number of steps allowed in recursion;
- a list of messages (used by the printing monad for debugging);
- the memory.

The size of the memory should ideally be dynamic, but at the same time the memory has to be statically typed. Our solution is to parametrize the memory by a signature Σ, containing the exact list of types:

$$T_1, T_2, \ldots, T_n$$

that will be used. The memory is a list of fixed size[7] of n regions respectively of types:

$$T_1, \ldots, T_n$$

A reference has type $\mathrm{Ref.t}\,\Sigma\,T_i$, and its implementation is simply the natural number i corresponding to the i-th type in the signature Σ. All in all, here are the effectful operations offered by the monad:

$$\begin{array}{rcl}
\mathrm{ref} & : & T_i \to M\,\Sigma\,(\mathrm{Ref.t}\,\Sigma\,T_i) \\
\mathrm{read} & : & \mathrm{Ref.t}\,\Sigma\,T \to M\,\Sigma\,T \\
\mathrm{write} & : & \mathrm{Ref.t}\,\Sigma\,T \to T \to M\,\Sigma\,() \\
\mathrm{print} & : & \alpha \to M\,\Sigma\,() \\
\mathrm{error} & : & \mathrm{string} \to M\,\Sigma\,\alpha \\
\mathrm{try_with} & : & (() \to M\,\Sigma\,\alpha) \to (\mathrm{string} \to M\,\Sigma\,\alpha) \to M\,\Sigma\,\alpha \\
\mathrm{dependentfix} & : & (\mathcal{F} \to \mathcal{F}) \to \mathcal{F} \text{ with } \mathcal{F} = \forall\,(x : A).\ M\,\Sigma\,(B\,x)
\end{array}$$

Pre-computation We partition the memory into the parts `InputMem` and the part `TmpMem`. `TmpMem` is initially empty and corresponds to the memory in the usual state monad. `InputMem` is initialized by the OCaml program and given as the initial (read-only) memory to Coq as a prophecy. Inside the implementation of the monad, we program differently for these two environments and, for this reason, we defined a low-level internal operator, `select`, of type:

$$\forall\,\alpha,\ (() \to \alpha) \to (() \to \alpha) \to \alpha$$

which is defined in Coq as:

$$\mathrm{select}(f, g) = f\,()$$

and compiled in OCaml as $\mathcal{C}(g())$. To fulfill the requirements to ensure that our monad is simulable, we make sure that the OCaml version of each operator only refines the contents of the `InputMem` during its effectful execution.

[7] Even if the list of types is finite, the memory may not be bounded if one the types holds an infinite number of values. Fox example, if T_1 is the type "list bool" then the size of the first memory cell may grow arbitrarily as the size of the list.

2.6.1 In OCaml

We implement the compilation of a monadic term written in Coq to a program in OCaml by customizing the existing extraction mechanism of Coq [40]. We extract the new monadic constructs as follows:

$$
\begin{array}{rcl}
M\,\Sigma\,\alpha & \mapsto & \alpha \\
\text{unit} & \mapsto & \texttt{fun}\ x \to x \\
\text{bind} & \mapsto & \texttt{fun}\ x\ f \to f\ x \\
\text{print} & \mapsto & \texttt{fun}\ x \to \text{print\_endline}\ x \\
\text{dependentfix} & \mapsto & \texttt{let rec}\ \text{fix}\ f = \texttt{fun}\ x \to \\
 & & \quad \text{incr\_nbsteps}\ (); f\ (\text{fix} f)\ x \\
 & & \texttt{in}\ \text{fix}\ f\ x \\
\text{error} & \mapsto & \texttt{fun}\ x \to \text{failwith}\ x \\
\text{try\_with} & \mapsto & \texttt{fun}\ f\ h \to \texttt{try}\ f\ () \\
 & & \quad \texttt{with}\ s \to h\ s \\
\text{tmp\_ref} & \mapsto & \texttt{fun}\ i\ v \to \text{ref}\ v \\
\text{input\_ref} & \mapsto & \texttt{fun}\ i\ v \to \text{register\_ref}\ i\ v \\
\text{read} & \mapsto & \texttt{fun}\ r \to !r \\
\text{write} & \mapsto & \texttt{fun}\ r\ v \to r := v
\end{array}
$$

Since we are using the built-in effectful evaluation mechanisms of OCaml, we convert our monad to the identity monad and we define the `bind` and `unit` combinators accordingly.

We implement the print and partiality monad with the standard print functions and exceptions.

For the fixpoint operator, we add some instrumentations to count the number of iterations in a global variable. The function `incr_nbsteps` increments a global counter by one each time we iterate in a recursive function defined by the `dependentfix` construct.

We implement the memory operators with OCaml's references. We divide the references into `tmp_ref` and `input_ref` references. The `tmp_ref` references are standard OCaml's references. The `input_ref` references are OCaml's references registered in an array. Thus, these references are not garbage collected at the end of the execution so that we can use them to produce the prophecy at the end of the execution. The `register_ref` function creates a new `input_ref` reference by creating a standard reference with the `ref` operator, and registering it in a global array.

2.6.2 Communication from OCaml to Coq

Once the execution of the OCaml code is done, we generate the prophecy for Coq. The prophecy contains two parts, the number of steps used for the recursions with the `dependentfix` operator, and the array of `input_ref` references.

The number of steps is easy to communicate from OCaml to Coq, as this is just a natural number. The array of `input_ref` references is harder to communicate, since we need to reify OCaml data into Coq terms. Notice that this is

not possible in general, for example for abstractions or for proof terms, since the extraction to OCaml erases too much information from the source term. Our solution is to provide an ad-hoc reification mechanism using binary trees: for every type T in the input memory signature, the user needs to provide a morphism between T and a binary tree.

2.7 Examples

We now show examples of Coq programs written using a simulable monad. The first example describes how to write effectful programs, while the second example illustrates how the performance of an algorithm is greatly improved by using compilation to OCaml and cross-stage memoization.

2.7.1 Congruence-Closure

The congruence-closure problem is about proving equality of two first-order terms, given a set of known equalities. It can be solved efficiently using the union-find algorithm [2]. We already know [19] a reflexive version of the algorithm, which is purely functional and proven correct. A large part of the code is devoted to prove termination and implementing functional arrays. We wrote this algorithm in our system using the partiality monad to avoid proving termination. We focus on the `Find` function on the Figure 2.6. At high level this function retrieves the representative u' of the equivalent class of u, along with a proof of the equality among u and u'. It iterates over a hash-table from expressions (`Index.t` in the code) to expressions, crawling the hash table until an element points to itself. If that is the case, then we reach the representative. The hash table also contains the proof of equality, which is used transitively to compute the resulting equality proof.

Programming with effects in Coq Proving termination of the algorithm is hard since it requires to maintain the invariant that the table is not cyclic. Luckily, we are exempt to do such proof, thanks to the dependentfix operator that allows for non-termination. The hash-table is a mutable structure with a read and a write operation. It is implemented as a mutable map from expressions to expressions, with an additional proof of equality.

Dependently-typed programming with partial functions We keep the power of the Coq type system despite the fact that we are working in a monad. The `Find` function has a dependent type specifying that the result is the representative term u' , equal to the input term u. The proof term is generated in the monad, so we can rely on run-time checks, which may fail, instead of proving invariants. For example the invariant $i = i'$ holds but does not have to be statically proven. Instead it is checked dynamically (comparison of i and i'

```
Program Definition Find hash u : M Σ {u' : Index.t | u ≡ u'} :=
  (* The dependentfix function creates a fixpoint and has type:
    ∀ {Σ A} (B : A → Type),
      ((∀ x, M Σ (B x)) → ∀ x, M Σ (B x)) →
      ∀ x, M Σ (B x)
  *)
  dependentfix (λ i ⇒ {j : Index.t | i ≡ j})(λ find i ⇒
    let! eq_proof := MHash.Read hash i in
    (* A cell of the hash contains the next element j and a proof of
      congruence to it. *)
    let (i', j, H_{i',j}) := eq_proof in
    if i ≡ i' then (* case i = i': should always be the case *)
      if i ≡ j then (* case i = j: we found it *)
        (* We use the proof H_{i',j} and the congruence of i and j
          to compute the proof required by "exist". *)
        return (exist _ j _)
      else (* case i <> j: we have to continue from j *)
        let! r := find j in
        let (k, H_{j,k}) := r in
        do! MHash.Write hash i
          (* To make the proof of equality between i and k we use the
            proofs H_{i',j} and H_{j,k} of the equalities i' = j and j = k and
            the equality of i and i'. We complete the proof in the
            proof mode of Coq thanks to the Program instruction.
          *)
          (EqProof.Make (i := i) (j := k) _) in
        return (exist _ k _)
    else (* case i <> i': unexpected *)
      error "Find : i = i'")
  u.
```

Figure 2.6: The `Find` function.

on line 5). The result is used to coerce a proof of $i' = j$ to $i = j$ (done automatically by the `Program` command in our example). If the check fails, we raise an exception handled by the partiality monad. In this way we can partially specify our programs. Notice that we are not forced to use partial programs, we can also use pure Coq functions leading to stronger static guarantees. This flexibility is not available in mainstream functional languages like OCaml.

In this example, to evaluate the `Find` function, we both use the prophecy as a termination certificate and computations in Coq in a state and partiality monad.

2.7.2 A tactic for Lattices

James and Hinze [35] present a reflection-based tactic to solve lattice equalities or inequalities based on the algorithm proposed by Whitman [64]. This algorithm is known for being exponential in the worst case. In this work, the authors made the following remark:

"Possible future work is to turn our current implementation [...] into one that uses dynamic programming to memoize the recursive calls. However, this is not a trivial task. Coq's programming language is purely functional [...], so any data-structure that we use for memoization must be purely functional and operations on that data-structure must all be proved terminating."

In this section we provide a tactic similar to James and Hinze's, but that uses memoization. In our case, unlike in the recommendation made in the quoted text, we use a form of cross-stage memoization to remember the successful path of execution made by the OCaml version and to transmit it to the Coq version. In this way, the exponential algorithm is executed only in OCaml, while Coq just recreates the successful path made by the OCaml version mimicking what a certificate checker would do. Unsurprisingly, the implementation presented here greatly outperforms the one presented by James and Hinze. For details, we refer to the original work cited above or to the accompanying code.

Whitman's algorithm We present the *Whitman's* algorithm, as written by James and Hinze, with some simplifications and syntax sugaring (Figure 2.7). What is important to notice is the $\vee$ branching in the last three cases. In particular, the last case requires the algorithm to branch four times! This is the culprit for the exponential time taken by the algorithm in some examples.

Remembering the past To simulate only the interesting part of the proof search, the simulation must choose the right side of every disjunction. This optimization lies on the function `tryBranches` on the Figure 2.8 which advantageously replaces $\vee$. This function uses the select operator to behave differently in OCaml than in Coq. In OCaml, it tries to execute the code in all of the branches, and the returned value comes from the first branch succeeding in its execution. In addition, the position of the successful branch is added to the map referenced by ref. If no branch succeed, then it raises an error. Before exploring

```
1  Program Fixpoint leq (t u : Term) : {b : bool | b → t ≤ u} :=
2    match (t, u) with
3    | (Var m, Var n) ⇒ m ≡ n
4    | (Join t1 t2, u) ⇒ leq t1 u ∧ leq t2 u
5    | (t, Meet u1 u2) ⇒ leq t u1 ∧ leq t u2
6    | (Var m, Join u1 u2) ⇒ leq t u1 ∨ leq t u2
7    | (Meet t1 t2, Var n) ⇒ leq t1 u ∨ leq t2 u
8    | (Meet t1 t2, Join u1 u2) ⇒ leq t1 u ∨ leq t2 u ∨ leq t u1 ∨ leq t u2
9    end.
```

Figure 2.7: The *Whitman's* algorithm.

```
1  Fixpoint tryBranches (ref : Ref.t Σ _) (n_branch : nat)
2    (k : TermPairMap.key) B (branches : list (unit → M Σ B))
3    : M Σ B :=
4    select
5    (* Coq *) (λ_ ⇒ let! map := !ref in
6      let! n_branch :=
7        extract_some (TermPairMap.find k map) in
8      let! branch :=
9        extract_some (nth_error branches n_branch) in
10     branch ())
11   (* OCaml *) (λ_ ⇒ let! map := !ref in
12     match TermPairMap.find k map with
13     | Some n ⇒
14       let! branch := extract_some (nth_error branches n) in
15       branch ()
16     | None ⇒ match branches with
17       | nil ⇒ error "No branch left to try"
18       | branch :: branches' ⇒ try!
19           let! r := branch () in
20           let! map := !ref in
21           do! ref := !TermPairMap.add k n_branch map in
22           return r
23         with _ ⇒ tryBranches ref (S n_branch) k branches'
24       end
25     end).
```

Figure 2.8: Function `tryBranches`.

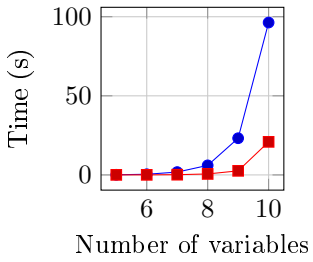

Figure 2.9: Typechecking of exponential proof terms.

the branches, it first checks whether it is known which branch to take, and if this is the case, it executes the code from that branch only. In Coq, it first reads the position from the map, and executes only the code from the branch in this position. In both cases, the key k used to store the position in the map is given as a parameter. This key is instantiated with a pair containing the terms from both sides of the inequality under consideration. The Whitman's algorithm is changed to take advantage of the branching function just described.

There is a couple of important remarks which must be made about this optimization. First, the powerful `select` operator can break the theoretical requirements to achieve the *a posteriori* simulation. As a result, a decision procedure could successfully generate a prophecy in OCaml but the Coq version could fail to execute, that is to say the final proof term would generate a type checking error. We provide the `select` operator to allow the user to create primitive operators not present in the monad. Second, our implementation of the non-determinism assumes that there is no side-effect in the failing branches that may affect the successful ones.

Performance As expected, we get a great performance gain, shown in Figures 2.9 and 2.10. These plots show the time it takes Coq to typecheck the result, for two different classes of problems. The time to typecheck the result from the original purely functional algorithm is shown in rounded dots, while for the effectful code it is shown in squares. In Figure 2.9 we consider a problem with an increasing number of variables, where there is no repetition in the formula (therefore every combination should be taken into account). In Figure 2.10, we increment the number of times a certain pattern occurs in an inequality, showing how our method benefits from reusing previously computed paths. To sum things up, these plots clearly shows the benefit of using prophecies to help the typechecker save some computation.

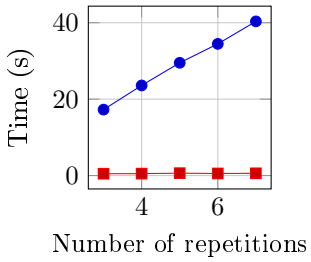

Figure 2.10: Typechecking time of repetitive terms.

2.8 Related work

Coq has been extended with imperative features [1]. The methodology behind this extension is to offer to Coq's user a functional interface to data structures that are efficiently compiled internally. This solution is transparent to the user: there is no need to write decision procedures in a monad to use imperative mechanisms. Yet, the trusted base, i.e. the kernel of Coq, had to be extended. Actually, the two systems can be used together: we could make use of the efficient data structures provided by this extension to define some of the effectful operators of our monad improving the performance of the a posteriori simulation done at Qed time.

Several works propose [5, 58] methods to define and to reason on general recursive functions in Type Theory. Bove and Capretta [5] formally define a notion of prophecy, a coinductive predicate derived from a set of non-overlaping recursive equations characterizing the co-domain of the partial function defined by these equations. Our prophecies and Bove and Capretta's share the same role of prediction. However, our prophecies do not need to be co-inductive because our monad uses them in direct style. Besides, our prophecies are computed outside Coq by an efficient computational model i.e. OCaml.

2.9 Conclusion

In this chapter, we presented a novel technique to write decision procedures in Coq. We described its implementation as a plugin and hope that it will simplify the development of proofs by reflection.

Part II

Importing imperative code in Coq

Chapter 3

CoqOfOCaml

The content of this chapter was presented at the OCaml 2014 *workshop.*

3.1 Abstract

There are hundreds of OCaml programs available on the Web. This represents a lot of real-world code written in a functional programming language with effects. On the other side, the Coq programming language has a much more limited programming ecosystem but features an expressive type system enabling the formal verification of important program properties.

We designed the CoqOfOCaml compiler to automatically import OCaml programs into equivalent Coq programs using an explicit effect system. These effects are automatically inferred from the OCaml programs. We support a large-enough subset of OCaml to import some modules of the standard OCaml library. Unlike the monolithic monad of Cybele, we chose a set of small composable monads in order to explicit the effects of each function.

3.2 Introduction

The OCaml language[1] is a functional programming language, with implicit side effects and a call-by-value evaluation strategy. This language is part of the ML language family [44], featuring a polymorphic type system with an automatic type inference algorithm [20]. At the time of this writing, a thousand of OCaml packages are available through the OPAM package manager[2].

The Coq language is a purely functional programming language with a dependently type system based on the CoC (Calculus Of Constructions) [18]. Thanks

[1] The OCaml language is available on ocaml.org.
[2] See the OCaml OPAM repository on opam.ocaml.org.

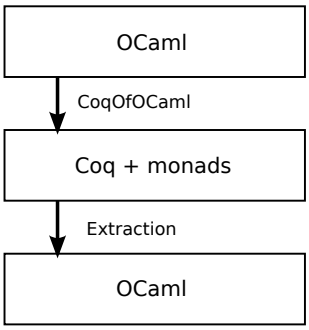

Figure 3.1: A typical workflow using the CoqOfOCaml compiler.

to this rich type system, we can formalize and prove correct complex mathematical theorems or program properties, like for example the soundness of a C compiler [39]. Unfortunately, far less programming libraries are available on the Coq platform[3]. Moreover, Coq being a purely functional language, side effects such as mutable references, exceptions, non-termination or inputs–outputs are forbidden, limiting the scope of the programs we can express.

In an attempt to provide a verification platform for the OCaml language, augment the number of programming libraries in Coq and understand what kinds of effect system are needed for practical programming, we developed the CoqOfOCaml compiler[4] to automatically import OCaml programs with side effects in the Coq language.

A typical workflow with the CoqOfOCaml compiler is presented on the Figure 3.1. First, we import an existing OCaml program to the Coq language, augmented with *monads* [63] to represent impure operations. We can edit the resulting program, add assertions and prove some properties. Then, using the standard extraction mechanism [40] of Coq, we can get back an OCaml runnable program.

In this chapter, we will present:

- a practical usage of the CoqOfOCaml compiler thought some running examples (page 49);
- an effect system including mutable references, exceptions, non-termination and inputs–outputs, together with an algorithm to automatically infer these effects from an OCaml program (page 59);
- an encoding of this effect system in the purely functional programming language Coq, with the use of a composable *sequential monad* (page 61);

[3] See a package list on coq.io/opam.

[4] The CoqOfOCaml compiler is written in OCaml and available under MIT license on github.com/clarus/coq-of-ocaml.

- some case studies with the import of the `List`, `Set` and `Map` modules of the standard OCaml library and a comparison between the size of the original and of the generated code (page 66).

3.3 Running examples

We will start by some examples to understand how the compiler CoqOfOCaml works.

3.3.1 Purely functional programs

The purely functional part of OCaml translates naturally to equivalent Coq programs. We handle λ-expressions, polymorphic definitions, constructors and pattern matching, records, type definitions (inductive types, records, synonyms, abstract types) and modules (without functors).

Let us take the following purely functional OCaml example:

```
1  let first x y = x
2
3  type ('a, 'b) sum =
4    | Inl of 'a
5    | Inr of 'b
6
7  let x =
8    match Inl 12 with
9    | Inl n -> first n false
10   | Inr _ -> 0
```

We define a polymorphic function `first` which returns its first argument (line 1) and an algebraic data type `sum` (line 3). Then, we define `x` as an expression which evaluates to the integer 12 (line 8). The CoqOfOCaml compiler generates the following Coq code:

```
1  Require Import OCaml.OCaml.
2
3  Local Open Scope Z_scope.
4  Local Open Scope type_scope.
5  Import ListNotations.
6
7  Definition first {A B : Type} (x : A) (y : B) : A := x.
8
9  Inductive sum (a b : Type) : Type :=
10 | Inl : a -> sum a b
11 | Inr : b -> sum a b.
12 Arguments Inl {a} {b} _.
13 Arguments Inr {a} {b} _.
```

```
14
15  Definition x : Z :=
16    match Inl 12 with
17    | Inl n => first n false
18    | Inr _ => 0
19    end.
```

Apart from a different syntax, this Coq code has the same shape as the original OCaml code. Indeed, Coq being a purely functional language, the pure constructs of OCaml have a direct equivalent in Coq. On line 1, we load the library `OCaml.OCaml` which contains some definitions for OCaml functions which are not present in the Coq standard library. Next, we activate some notations (lines 3 to 5). We explicit the type parameters A and B of the polymorphic function `first` (line 7), because they would not be inferred by Coq. We declare them with curly braces so that they can be inferred when the function `first` is applied.

We devoted some efforts to automatically generate of a human-readable code, so that the Coq code can be edited afterward and not just compiled. This led to the creation of the pretty-printing library SmartPrint[5].

3.3.2 Programs with effects

Since Coq is a purely functional language, effects from OCaml programs must be encoded in Coq. We represent effects with a *monad* [63], a structure representing a sequence of effectful operations. We encode effectful programs returning a value of type A, with a state of type S and a potential exception of type E, by the type:

$$\mathcal{M}_{S,E}\, A := S \to (A + E) \times S$$

Single reference To simplify the static analysis, we only consider global mutable references[6]. The following OCaml code:

```
1  let n = ref 0
2
3  let incr () =
4    n := !n + 1
```

compiles to:

```
1  Definition n := Effect.make Z Empty_set.
2
3  Definition read_n (_ : unit) : M [ n ] Z :=
4    fun s => (inl (fst s), s).
```

[5] The SmartPrint library is written in OCaml and available under BSD license on github.com/clarus/smart-print.

[6] The use of an unrestricted form of references would require the design of a parametrized effect system, which outreaches the scope of this chapter and did not implement.

```
5
6   Definition write_n (x : Z) : M [ n ] unit :=
7     fun s => (inl tt, (x, tt)).
8
9   Definition incr (x : unit) : M [ n ] unit :=
10    match x with
11    | tt =>
12      let! x_1 :=
13        let! x_1 := read_n tt in
14        ret (Z.add x_1 1) in
15      write_n x_1
16    end.
```

We declare the reference n (line 1) as an effect with a state of type $\mathbb{Z}$ (the type of the integers) and an error of type $\emptyset$ (the empty set, since standard reference operations cannot raise exceptions). The operations `read_n` and `write_n` (line 3 to 7) are defined to read and write the reference n. The result type of `read_n` is:

$$\mathcal{M}\,[n]\,\mathbb{Z}$$

meaning that `read_n` is an impure operation of effect n with a result of type $\mathbb{Z}$. The function `write_n` writes in a state of type $\mathbb{Z} \times \texttt{unit}$ instead of $\mathbb{Z}$ for technical reasons, to simplify recurrences when using a list of references. To define the `incr` function (line 9 to 16), we use the monadic *bind*[7]:

$$\texttt{let! } x := e_1 \texttt{ in } e_2$$

which sequences the effectful expression e_1 with the effectful expression e_2 in which the variable x is bound to the result of e_1. The *return* operator:

$$\texttt{ret } e$$

lifts a pure expression e to an effectful expression `ret` e.

Multiple references To import an OCaml code with multiple references, we need to compose different effects. For example, for this Coq code with two references:

```
1   let n = ref 0
2   let m = ref 1
3
4   let swap () =
5     let tmp = !n in
6     n := !m;
7     m := tmp
```

[7] A monad is a structure with two operators:

$$\begin{cases} \text{return} : \alpha \to \mathcal{M}\,\alpha \\ \text{bind} : \mathcal{M}\,\alpha \to (\alpha \to \mathcal{M}\,\beta) \to \mathcal{M}\,\beta \end{cases}$$

we get the following Coq code:

```
1  Definition swap (x : unit) : M [ n; m ] unit :=
2    match x with
3    | tt =>
4      let! tmp := lift [_;_] "10" (read_n tt) in
5      let! _ :=
6        let! x_1 := lift [_;_] "01" (read_m tt) in
7        lift [_;_] "10" (write_n x_1) in
8      lift [_;_] "01" (write_m tmp)
9    end.
```

The return type of the `swap` function is:

$$\mathcal{M}\,[n; m]\,\text{unit}$$

meaning that the `swap` function uses the two reference effects n and m. In order to mix subexpressions manipulating the reference n with subexpressions manipulating the reference m, we introduce the `lift` function. On line 4:

$$\texttt{lift}\ [\_\,;\,\_]\ \text{"10"}\ (\texttt{read\_n}\ \texttt{tt})$$

is of type:

$$\mathcal{M}\,[n; m]\,\mathbb{Z}$$

but `read_n tt` without the `lift` is of type $\mathcal{M}\,[n]\,\mathbb{Z}$. We lift each subexpression to the set of effects used in its context, so that all the subexpressions of an expression appear to have the same effects. In order to keep the syntax of the `lift` short and inferable, we use a syntax trick:

$$\text{lift}\,l\,s$$

with l a list of k underscores for k names of effects to infer and s a string of k zeros and ones, acting as a mask to select the effects to introduce in the lift.

Exceptions CoqOfOCaml imports OCaml codes containing exceptions. For example:

```
1  exception Error of string
2
3  let n =
4    try raise (Error "failure") with
5    | Error _ -> 12
```

is imported to the following Coq code:

```
1  Definition Error := Effect.make unit (string).
2
3  Definition raise_Error {A : Type} (x : string)
```

```
4     : M [ Error ] A :=
5     fun s => (inr (inl x), s).
6
7   Definition n : Z :=
8     match Exception.run 0 (raise_Error ("failure")) tt with
9     | inl x => x
10    | inr (_) => 12
11    end.
```

The effect `Error` is defined on line 1 as the effect with an empty state (of type `unit`) and an error with a `string` payload. The associated `raise_Error` function (line 3 to 5) raises an exception of kind `Error` and has the return type:

$$\mathcal{M}\,[\text{Error}]\,\alpha$$

for any type α. Indeed, since the raising of an exception does not return a value, the return type of `raise_Error` can be any type α[8]. The definition of n (line 7 to 11) successfully catches the exception so that n is a pure value of type $\mathbb{Z}$. To catch the exception, we use the function `Exception.run` 0 which transforms the first exception (hence the index 0) in the list of effects (here, we only have the exception `Error`) to a sum type.

Non-termination Function which may not terminate cannot be represented in Coq for consistency reasons[9]. We encode these functions using the technique of the fuel: a counter acting for a fuel is decremented at each iteration. We raise an exception if this counter reaches zero. Thus, the recursive functions may fail but we are sure they terminate within a bounded number of steps. The non-termination effect combines a mutable reference (the fuel counter) and an exception.

The standard `map` function of OCaml:

```
1   let rec map f l =
2     match l with
3     | [] -> []
4     | x :: l -> f x :: map f l
5
```

is compiled to Coq using the non-termination effect:

```
1   Fixpoint map_rec {A B : Type} (counter : nat) (f : A -> B)
2     (l : list A) : M [ NonTermination ] (list B) :=
3     match counter with
```

[8] In OCaml, the type of the function `raise` is also exn $\to \alpha$.

[9] With the non-termination, we could easily prove `False`:

$$\texttt{Fixpoint}\ f : \text{False} := f.$$

```
4     | O => not_terminated tt
5     | S counter =>
6       match l with
7       | [] => ret []
8       | cons x l =>
9         let! x_1 := (map_rec counter) f l in
10        ret (cons (f x) x_1)
11      end
12    end.
13
14  Definition map {A B : Type} (f : A -> B) (l : list A)
15    : M [ Counter; NonTermination ] (list B) :=
16    let! x := lift [_;_] "10" (read_counter tt) in
17    lift [_;_] "01" (map_rec x f l).
```

This code contains two functions:

- `map_rec`, a terminating function defined by induction over the fuel counter. The `map_rec` function may raise an exception if the counter reaches zero (line 4);
- `map`, the wrapping function which create the fuel counter with a mutable reference `Counter`.

The `map` function is annotated with two effects, the mutable reference `Counter` and the exception `NonTermination`. Since the `map` function is defined by induction over the list l, it would be preferable to let Coq guess its termination. We can also do so by adding the `coq_rec` tag to the OCaml definition[10]:

```
1  let rec map f l =
2    match l with
3    | [] -> []
4    | x :: l -> f x :: map f l
5  [@@coq_rec]
```

which yields the simpler Coq code:

```
1  Fixpoint map {A B : Type} (f : A -> B) (l : list A)
2    : list B :=
3    match l with
4    | [] => []
5    | cons x l => cons (f x) (map f l)
6    end.
```

Inputs–outputs A special effect `IO`[11] represents the interactions with the outer world. For this *Hello world* program in OCaml:

[10] For local fixpoints, you have to suffix the function name by `_coq_rec`.
[11] The name IO is a reference to the the IO monad of Haskell.

```
1  let main () =
2    print_endline "Hello world!"
```

we get the following Coq program:

```
1  Definition main (x : unit) : M [ IO ] unit :=
2    match x with
3    | tt => OCaml.Pervasives.print_endline "Hello world!"
4    end.
```

In Coq, the `IO` effect is encoded as a mutable reference to a value representing the current world state. We remark that the OCaml function `print_endline` is imported to a Coq function:

`OCaml.Pervasives.print_endline`

defined in the `OCaml.OCaml` module. Many functions in the `Pervasives` and the `List` modules of OCaml are recognized by the CoqOfOCaml compiler and imported in the same way.

3.4 Intermediate language

We will present the intermediate language $\mathcal{L}$ used by the compiler CoqOfOCaml to represent and transform OCaml programs to Coq programs, through the steps given on the Figure 3.2.

3.4.1 The language $\mathcal{L}$

The language $\mathcal{L}$ first serves to import an OCaml program to an AST (Abstract Syntax Tree). To do so, we reuse the parser and the type checker of the OCaml compiler through the `compiler-libs` library. We define an element of $\mathcal{L}$ as a list of structures.

3.4.2 Structure

The top-level of an OCaml file is a list of definitions of values, types or modules. These top-level definitions are encoded by the type $\mathcal{S}$ (for *structure*, following the OCaml's naming), defined in the Figure 3.3. The type $\mathcal{S}$ is parametrized by a type of annotations α, which we will use in particular to add effect annotations. In order to produce readable error messages, each node of a structure is also annotated by a line number in the implementation.

We handle six kinds of structure items:

- the definition of a value (definitions with the `let` keyword);

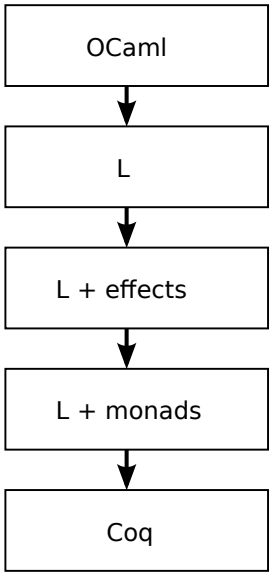

Figure 3.2: Compilation steps of the compiler.

$$
\begin{array}{rcl}
\mathcal{S}\,\alpha & := & \mathrm{Value}\,\alpha \\
& | & \mathrm{TypeDefinition} \\
& | & \mathrm{Exception} \\
& | & \mathrm{Reference} \\
& | & \mathrm{Open} \\
& | & \mathrm{Name} \times \mathrm{list}\,(\mathcal{S}\,\alpha)
\end{array}
$$

Figure 3.3: The annotated language of structures.

$$\begin{cases} \text{name} & : & \text{Name} \\ \text{typ\_vars} & : & \text{list Name} \\ \text{args} & : & \text{list (Name} \times \text{Type)} \\ \text{typ} & : & \text{option Type} \\ \text{body} & : & \mathcal{E}\,\alpha \end{cases}$$

Figure 3.4: A case in a value definition.

- the definition of a type;
- the definition of a kind of exception;
- the definition of a global reference (which is, in OCaml, viewed as a particular case of definition of value);
- the opening of a module;
- the definition of a module, which is a list of structures.

3.4.3 Values

The values are the most complex kind of structure items. They are a list of potentially mutually-recursive definition cases. Each definition case is described by the list of fields given in the Figure 3.4. A case contains the name of a new value, a list of polymorphic type variables, a list of typed arguments (empty for the definition of ground values), an optional return type and the body of the definition.

The body of the definition is an annotated expression of type $\mathcal{E}\,\alpha$. The expressions are described in the Figure 3.5. Each expression node is annotated by an element of type α. Most kinds of nodes are self-explanatory and reflect OCaml constructs. Some special nodes are introduced to represent explicit effect operators, namely `Return`, `Bind`, and `Lift`. We will present these effect operators page 61. The `Run` node represents the `try-with` operator. We distinguish the `LetVar` (for definitions with no arguments) from the more general `LetFun` construct (for definitions with arguments) in order to simplify the implementation. Indeed, definitions with no arguments will require a different treatment from the definitions with at least one argument. Similarly, the `Tuple`, `IfThenElse` and `Sequence` constructs could be viewed as special cases of the `Constructor`, `Match` and `Bind` constructs (respectively), but we prefer to keep them to maintain the original structure of the compiled OCaml programs.

The type `Name` represents new names, while the type `BoundName` stands for names defined in the current context. A `BoundName` is described by a base name, a module path and a context depth, to prevent a bound name to become

$$
\begin{array}{rcl}
\mathcal{E}\,\alpha & := & \text{Constant}(\alpha \times \text{Constant}) \\
& | & \text{Variable}(\alpha \times \text{BoundName}) \\
& | & \text{Tuple}(\alpha \times \text{list}\,(\mathcal{E}\,\alpha)) \\
& | & \text{Constructor}(\alpha \times \text{BoundName} \times \text{list}\,(\mathcal{E}\,\alpha)) \\
& | & \text{Apply}(\alpha \times \mathcal{E}\,\alpha \times \text{list}\,(\mathcal{E}\,\alpha)) \\
& | & \text{Function}(\alpha \times \text{Name} \times \mathcal{E}\,\alpha) \\
& | & \text{LetVar}(\alpha \times \text{Name} \times \mathcal{E}\,\alpha \times \mathcal{E}\,\alpha) \\
& | & \text{LetFun}(\alpha \times \text{Definition}\,\alpha \times \mathcal{E}\,\alpha) \\
& | & \text{Match}(\alpha \times \mathcal{E}\,\alpha \times (\text{Pattern} \times \text{list}\,(\mathcal{E}\,\alpha))) \\
& | & \text{Record}(\alpha \times \text{list}\,(\text{BoundName} \times \mathcal{E}\,\alpha)) \\
& | & \text{Field}(\alpha \times \mathcal{E}\,\alpha \times \text{BoundName}) \\
& | & \text{IfThenElse}(\alpha \times \mathcal{E}\,\alpha \times \mathcal{E}\,\alpha \times \mathcal{E}\,\alpha) \\
& | & \text{Sequence}(\alpha \times \mathcal{E}\,\alpha \times \mathcal{E}\,\alpha) \\
& | & \text{Return}(\alpha \times \mathcal{E}\,\alpha) \\
& | & \text{Bind}(\alpha \times \mathcal{E}\,\alpha \times \text{option Name} \times \mathcal{E}\,\alpha) \\
& | & \text{Lift}(\alpha \times \text{Descriptor} \times \text{Descriptor} \times \mathcal{E}\,\alpha) \\
& | & \text{Run}(\alpha \times \text{BoundName} \times \text{Descriptor} \times \mathcal{E}\,\alpha)
\end{array}
$$

Figure 3.5: The annotated language of expressions.

$$
\begin{array}{l}
\text{TypeDefinition} := \\
\quad | \quad \text{Inductive}(\text{Name} \times \text{list Name} \times \text{list}\,(\text{Name} \times \text{list Type})) \\
\quad | \quad \text{Record}(\text{Name} \times \text{list}\,(\text{Name} \times \text{Type})) \\
\quad | \quad \text{Synonym}(\text{Name} \times \text{list Name} \times \text{Type}) \\
\quad | \quad \text{Abstract}(\text{Name} \times \text{list Name})
\end{array}
$$

Figure 3.6: The type definitions.

inaccessible in case of redefinition[12].

3.4.4 Type definitions

A type definition can have one of the four forms described in the Figure 3.6. An inductive definition contains a list of type parameters and a list of constructors. We do not check for the strict positivity of the constructors and instead rely on the Coq type checker. A record is a list of named fields. A type synonym may contain some type parameters. Finally, an abstract type is a type whose definition is hidden. We compile it to an axiom in Coq.

[12] This permits to type x in:

```
type t = A
module M = struct
  type t = B
  let x = A
end
```

3.4.5 Exceptions and references

Exceptions are defined by a name and a single type of argument. Likewise, global references are defined by a name and the type of the value pointed by the reference.

3.5 Effects inference

ONCE A PROGRAM is imported to the language $\mathcal{L}$, we infer its implicit effects and annotate it. We will describe the effect system of the compiler CoqOfOCaml and its inference algorithm.

3.5.1 Effects

We restricted ourselves to a static effect system with no parametrized effects, keeping this extension as a future work. To parametrize the effects, we could for example follow the model of the language Koka [38], but this would complexify our implementation.

Definition 2 (Effect identifier)**.** *An* effect identifier *is a global name denoting an effect. An effect identifier can denote a reference, an exception or the special IO effect.*

Definition 3 (Effect descriptor)**.** *An* effect descriptor *is a set of effect identifiers.*

We do not enforce any order on the effect identifiers, so that there are no differences between a function using a reference *and* raising an exception, and a function raising an exception *and* using a reference[13]. We compose two effect descriptors using the set union.

Definition 4 (Effect type)**.** *An* effect type *is the shape of an OCaml type with additional effect information. The syntax is the following:*

$$\begin{array}{rcl} \tau & := & \text{Pure} \\ & | & \text{Pure} \xrightarrow{d} \tau \end{array}$$

A value can be pure (as effect-free and terminating, even when it is a function which we apply) or can be a function using the effects defined by the effect descriptor d when applied. All function arguments are effects-free. This restriction avoids the need of a parametrized effect system. However, this forbids the definition of some higher-order imperative functions, such as:

$$\texttt{List.iter} : (\alpha \to \text{unit}) \to \text{list}\,\alpha \to \text{unit}$$

[13] This contrasts with the behavior of the *monad transformers*, which would not be commutative in this case.

Indeed, with our effect system, the first argument of `List.iter` must be a pure function where we expected an effectful function. We could handle such higher-order functions with effects using a more expressive effect system (such as the one in Koka [38]).

Definition 5 (Effect)**.** *An* effect *is a couple of an effect descriptor and an effect type.*

We use an effect to describe the behavior of an expression. Indeed, contrary to a value, an expression can also generate effects during its evaluation.

3.5.2 Inference

The effect inference is the process of annotating each subexpression of a program by its effect:

$$\text{infer} : \mathcal{L}\,\text{unit} \to \mathcal{L}\,\text{effect}$$

To define the `infer` function, we process by structural induction over the tree of a program, keeping the effect type of the values defined in the current context. For (mutually) recursive functions, we compute a fixpoint of the effects until convergence[14]. This fixpoint must terminate because:

- there is a finite number of declared effects for a given context;
- the size of the effect type of a value is bounded by the size of its type.

As an example, we can infer the effects of the following function:

```
1  let rec prints l =
2    match l with
3    | [] -> ()
4    | s :: l ->
5      print_endline s;
6      prints l
7  [@@coq_rec]
```

which prints a list of strings, in a context where `print_endline` has the effect type:

$$\text{print\_endline} \quad :: \quad \text{Pure} \xrightarrow{\mathsf{IO}} \text{Pure}$$

In the body of the function (line 2 to 6), we add the pure parameter l and the `prints` function itself, which we suppose pure to start with:

$$\begin{aligned} l &:: \text{Pure} \\ \text{prints} &:: \text{Pure} \end{aligned}$$

The first branch of the `match` (line 3) is pure. The second branch (lines 4 to 6) applies the pure function `prints` and the impure function `print_endline`, generating the IO effect. We conclude that:

$$\text{prints} \quad :: \quad \text{Pure} \xrightarrow{\mathsf{IO}} \text{Pure}$$

[14] In practice, this fixpoint usually converges in one step.

By the same reasoning, supposing that `prints` is impure, we compute the same effect type:

$$\text{prints} \quad :: \quad \text{Pure} \xrightarrow{\mathsf{IO}} \text{Pure}$$

Thus our fixpoint is reached and we have computed the effect type of `prints`.

3.6 Monadic transformation

We need to encode the OCaml effects in Coq since Coq is a purely functional language. We will use a composable form of monad, parametrized by the effect descriptors.

3.6.1 Sequential monad

Definition 6 (Sequential monad)**.** *For a state type S and an error type E, we define the* sequential monad $\mathcal{M}_{S,E}$ *as:*

$$\begin{array}{lcl} \mathcal{M}_{S,E}\,\alpha & = & S \to (\alpha + E) \times S \\ \text{return}\, x & = & \lambda s.\, (\texttt{inl}\, x, s) \\ \text{bind}\, x\, f & = & \lambda s.\, \texttt{match}\ x\ s\ \texttt{with} \\ & & \quad \mid (\texttt{inl}\, v, s') \Rightarrow f\, v\, s' \\ & & \quad \mid (\texttt{inr}\, e, s') \Rightarrow (\texttt{inr}\, e, s') \\ & & \quad \texttt{end} \end{array}$$

We remark that each of our effects (references, exceptions and IO) can be represented by the monad $\mathcal{M}_{S,E}$, for some well-chosen types S and E. We associate each effect identifier i to a state type S_i and an error type E_i. For a reference to a value of type T we take:

$$S = T; E = \emptyset$$

For an exception of parameter T:

$$S = \text{unit}; E = T$$

For the IO effect:

$$S = \text{World}; E = \emptyset$$

for some axiomatized `World` type.

We chose an arbitrary total order on the effect identifiers, so that an effect descriptor d can be represented as its unique ordered list of effect identifiers:

$$d = [i_1, \ldots, i_n] \quad \text{with} \quad i_1 < \cdots < i_n$$

Definition 7 (Parametrized monad)**.** *For an effect descriptor $d = [i_1, \ldots, i_n]$, we define the monad $\mathcal{M}_d$ as:*

$$\mathcal{M}_d \quad = \quad \mathcal{M}_{S_d, E_d}$$

with:

$$\begin{cases} S_d &= S_{i_1} \times \cdots \times S_{i_n} \\ E_d &= E_{i_1} + \cdots + E_{i_n} \end{cases}$$

This composition is commutative[15] by construction. Said otherwise, the state of a set of effect identifiers is the product of its states, and the error of a set of effect identifiers is the sum of its errors. We define the composition of two monads $\mathcal{M}_{d_1}$ and $\mathcal{M}_{d_2}$ as the monad $\mathcal{M}_{d_1 \cup d_2}$.

Definition 8 (Lift)**.** *For two effect descriptors d and d' with $d \subseteq d'$, we define a `lift` operator:*

$$\begin{array}{rcccl} \mathrm{lift}_{d,d'} & : & \mathcal{M}_d & \to & \mathcal{M}_{d'} \\ & & x & \mapsto & \lambda s'.\,\texttt{match}\ x\ \lfloor s' \rfloor\ \texttt{with} \\ & & & & \quad \mid (\texttt{inl}\, v, s) \Rightarrow (\texttt{inl}\, v, s \sqcup \lfloor s' \rfloor') \\ & & & & \quad \mid (\texttt{inr}\, e, s) \Rightarrow (\texttt{inr}\, \lceil e \rceil, s \sqcup \lfloor s' \rfloor') \\ & & & & \quad \texttt{end} \end{array}$$

using the canonical operators:

$$\begin{cases} \lfloor \rfloor & : & S_{d'} \to S_d \\ \lfloor \rfloor' & : & S_{d'} \to S_{d'-d} \\ \sqcup & : & S_d \to S_{d'-d} \to S_{d'} \\ \lceil \rceil & : & E_d \to E_{d'} \end{cases}$$

3.6.2 Transformation

We use the effect annotations to explicit the effects in Coq, by introducing the monadic operators `return`, `bind` and `lift`. Doing so, we transform annotated programs to programs without effect annotations:

$$\varphi : \mathcal{L}\,\mathrm{effect} \to \mathcal{L}\,\mathrm{unit}$$

We proceed by induction on each node of a program tree, doing a standard monadic transformation[16] with some lifting operations. For a given expression, if the subexpressions have a different effect descriptor, we know that by construction these effect descriptors must be included in the effect descriptor of the node. Thus we add a corresponding `lift` operation, so that the subexpressions always have the same effect descriptor as their enclosing expressions, and the applications of the `bind` operator are well-typed. In order to keep the

[15] Using the monad transformers, there are two ways to compose a reference and an exception:

$$\begin{cases} \mathcal{M}_1\,\alpha &= S \to (\alpha + E) \times S \\ \mathcal{M}_2\,\alpha &= (S \to \alpha \times S) + E \end{cases}$$

Our composition always leads to the first solution, which is what most programmer should expect and what happens in OCaml.

[16] See the lecture of Xavier Leroy on monadic transformations on xavierleroy.org/mpri/2-4/monads.pdf.

$$
\begin{array}{rcl}
\varphi\, c & = & c \\
\varphi\, x & = & x \\
\varphi\,(\lambda x.\, e) & = & \lambda x.\,(\varphi\, e) \\
\varphi\,(\texttt{let}\, x = e_1\ \texttt{in}\ e_2) & = & \mathrm{bind}_{d_1,d_2,d}\, x\,(\varphi\, e_1)\,(\varphi\, e_2) \\
 & & \text{with } e_1 :: d_1;\ e_2 :: d_2;\ d = d_1 \cup d_2 \\
\mathrm{bind}_{d_1,d_2,d}\, x\, e_1\, e_2 & = & \begin{cases} \texttt{let}\, x = e_1\ \texttt{in}\ e_2 & (\text{if } d_1 = \emptyset) \\ \mathrm{bind}\,(\mathrm{lift}_{d_1,d}\, e_1)\,(\lambda x.\,(\mathrm{lift}_{d_2,d}\, e_2)) \end{cases} \\
\mathrm{lift}_{d_1,d_2}\, e & = & \begin{cases} e & (\text{if } d_1 = d_2) \\ \mathrm{return}\, e & (\text{if } d_1 = \emptyset) \\ \mathrm{lift}\, d_1\, d_2\, e \end{cases}
\end{array}
$$

Figure 3.7: Some rules of the monadic transformation.

generated code readable, we avoid to transform the purely functional expressions. We present on the Figure 3.7 some typical rules of this transformation. For a more complete and detailed description, you can look at the source code of CoqOfOCaml[17].

3.7 Generation of Coq

Once the monadic transformation is done, the code is ready to be pretty-printed using the syntax of Coq. We just need to implement the monad and the monadic operators in Coq.

We define an effect identifier as a couple of a state and an error type:

```
1  Record Identifier.t := {
2    S : Type;
3    E : Type }.
```

We encode an effect descriptor as a list of effect identifiers. We define the monad $\mathcal{M}_d$ by:

```
1  Definition M (es : list Identifier.t) (A : Type) : Type :=
2    match es with
3    | [] => A
4    | _ => state es -> (A + error es) * state es
5    end.
```

We define this monad like in the definition 7, with the exception of the empty case line 3. Due to technical simplifications, when the effect descriptor is empty, we consider that the monad is the purely functional monad.

By induction over the list of effect identifiers, we define various functions including the following operators:

[17] Available on github.com/clarus/coq-of-ocaml.

```
Definition lift {A : Type} (es : list Effect.t) (bs : string)
  (x : M _ A) : M _ A :=
  let aux (ebs : list (Effect.t * bool))
    (x : M (Effect.Ebs.sub ebs) A) : M (Effect.Ebs.domain ebs) A :=
    fun s =>
      let (r, s') := x (Effect.Ebs.filter_state ebs s) in
      let s := Effect.Ebs.expand_state ebs s' s in
      match r with
      | inl x => (inl x, s)
      | inr err => (inr (Effect.Ebs.expand_exception ebs err), s)
      end in
  let fix bool_list (s : string) : list bool :=
    match s with
    | EmptyString => []
    | String "0" s => false :: bool_list s
    | String _ s => true :: bool_list s
    end in
aux (List.combine es (bool_list bs)) x.
```

Figure 3.8: Definition of the `lift` function in Coq.

```
ret {es : list Identifier.t} {A : Type}
  (x : A) : M es A
bind {es : list Identifier.t} {A B : Type}
  (x : M es A) (f : A -> M es B) : M es B
lift {A : Type} (es : list Identifier.t) (bs : string)
  (x : M _ A) : M _ A
run {A : Type} {es : list Identifier.t} (n : nat)
  (x : M es A) (tt' : nth_is_stateless es n)
  : M (remove_nth es n)
    (A + Identifier.E (List.nth n es Identifier.nil))
```

The definitions of the `ret` and the `bind` operators are as expected, but the definitions of the `lift` and the `run` are more involving (Figure 3.8 and 3.9). For the `lift` operator, we use the trick of the string presented in the running example page 51. For the `run` operator, implementing the catching of an exception, we use a special argument of type:

$$\text{nth\_is\_stateless}\ es\ n$$

which is equal to the type `unit` if the n-th effect identifier of es has a state reduced to the trivial state `unit`. Thus, by calling the `run` operator with the `unit` value `tt`, we force the Coq type checker to verify that the n-th effect identifier is indeed a pure exception, otherwise the `run` could not be defined.

```
1  Fixpoint remove_nth (es : list Effect.t) (n : nat) : list Effect.t :=
2    ...
3
4  Definition nth_is_stateless (es : list Effect.t) (n : nat) : Type :=
5    match List.nth_error es n with
6    | Some e => Effect.S e
7    | None => unit
8    end.
9
10 Fixpoint input (es : list Effect.t) (n : nat)
11   (tt' : nth_is_stateless es n) (s : Effect.state (remove_nth es n))
12   {struct es} : Effect.state es.
13   ...
14
15 Fixpoint output (es : list Effect.t) (n : nat) (s : Effect.state es)
16   {struct es} : Effect.state (remove_nth es n).
17   ...
18
19 Fixpoint error (es : list Effect.t) (n : nat) (err : Effect.error es)
20   {struct es}
21   : Effect.E (nth n es Effect.nil) + Effect.error (remove_nth es n).
22   ...
23
24 Definition run {A : Type} {es : list Effect.t} (n : nat)
25   (x : M es A) (tt' : nth_is_stateless es n)
26   : M (remove_nth es n) (A + Effect.E (List.nth n es Effect.nil)) :=
27   of_raw (fun s =>
28     let (r, s) := (to_raw x) (input _ _ tt' s) in
29     (match r with
30     | inl x => inl (inl x)
31     | inr err => sum_assoc_left (inr (error _ _ err))
32     end, output _ _ s)).
```

Figure 3.9: Definition of the `run` function to catch an exception.

3.8 Specification

THE SPECIFICATION of the compiler CoqOfOCaml is twofolds. First, we require that an imported Coq program `foo.v` behaves as its original OCaml program `foo.ml` when we interpret its effects. By the same behavior we mean the same traces of inputs–outputs. Second, we require that the OCaml program `foo2.ml` extracted [40] from the Coq program `foo.v` behaves as the original `foo.ml` too.

We do not formally verify this specification. The verification of CoqOfOCaml would require a signification amount of work, because we would need to formalize the semantics of OCaml and to verify the extraction mechanism of Coq (there is already some work in this direction [25], but the extraction mechanism was not verified with the actual implementation of Coq).

3.9 Case studies

WE SUCCESSFULLY imported the slightly modified `List`, `Set` and `Map` modules from the standard OCaml library. These modules work on immutable structures but are not purely functional: they contain exceptions and functions whose termination is not obvious.

Here is an example of the `map2` function as defined in the `List` library of OCaml:

```
let rec map2 f l1 l2 =
  match (l1, l2) with
  | ([], []) -> []
  | (a1::l1, a2::l2) ->
    let r = f a1 a2 in r :: map2 f l1 l2
  | (_, _) -> invalid_arg "List.map2"
  [@@coq_rec]
```

Here is the imported version:

```
Fixpoint map2 {A B C : Type} (f : A -> B -> C)
  (l1 : list A) (l2 : list B)
  : M [ OCaml.Invalid_argument ] (list C) :=
  match (l1, l2) with
  | ([], []) => return []
  | (cons a1 l1, cons a2 l2) =>
    let r := f a1 a2 in
    let! x := map2 f l1 l2 in
    return (cons r x)
  | (_, _) =>
    OCaml.Pervasives.invalid_arg "List.map2" % string
end.
```

Module	Lines of OCaml	Lines of Coq	Increase
List	396	622	+57%
Set	349	539	+54%
Map	310	497	+60%

Table 3.1: Number of lines of the generated code.

The return type in Coq explicitly mentions that `map2` may raise the following exception:

```
OCaml.Invalid_argument
```

and that this is the only effect the `map2` function can do.

In the Table 3.1, we compare the number of lines of code of the original OCaml files to the number of lines in the generated Coq files. The size increase of around 60% was mainly due to the monadic translation, especially for the non-termination effect for which we have to define an auxiliary function. We hope that this size increase is small enough so that the users can continue to work easily on the imported Coq files.

We have not worked on the reasoning rules about the generated monadic programs yet. Still, it was possible to manually make a Coq proof about an absence of exception in the generated code. In the `List` module, the `sort` function depends on an auxiliary function `chop` which may raise an exception `Assert_failure`. Its importation into Coq is:

```
1  Fixpoint chop {A : Type} (k : Z) (l : list A)
2    : M [ OCaml.Assert_failure ] (list A) :=
3    ...
```

It removes the first k elements of a list l, and fails if there is no element left to remove. The precondition $0 \leq k \leq \text{length}\, l$ ensures that `chop` will succeed. We added this precondition in Coq and proved that the failing branch is never reached. Hence, we transformed `chop` into an effects-free function:

```
1  Fixpoint chop {A : Type} (k : Z) (l : list A)
2    {struct l} : 0 <= k -> k <= length l ->
3    { l' : list A | length l' = length l - k }.
```

Using this new `chop` function, we updated the definition of `sort` and proved that it never raises an exception.

3.10 Related work

OTHER TOOLS have been developed to formally verify functional programs with effects.

The CFML [11] project provides a compiler to import an OCaml program with effects in Coq, by representing it with a logical formula, the *characteristic formula*. We can prove properties about the characteristic formula using the CFML library of Coq tactics. Contrary to the CFML compiler, CoqOfOCaml generates a shallow embedding of an OCaml program. We believe that the shallow embeddings are more practical to prove program properties, because we can reuse existing Coq tools such as the Program plugin [59]. Moreover, importing existing OCaml libraries in the Coq language is useful for the Coq programmer, since less programming libraries are available in the Coq ecosystem.

The Ynot [47] system defines a monad to do imperative programming with memory pointers in Coq and to prove properties using Hoare logics [32]. This monad is more low-level than ours, and can express complex pointer manipulations where we only provide global mutable references. An interesting project would be to design an automatic translation from the monad of CoqOfOCaml to the monad of Ynot, in order to use their reasoning techniques to verify existing OCaml programs.

The Why3 [23] platform provides an impure functional programming language WhyML which can be compiled to OCaml. The language WhyML includes an annotation mechanism to write assertions and verification conditions generator. These conditions can be translated to various theorem provers including Coq.

There is also a compiler hs-to-gallina[18] which imports programs written in Haskell into Coq. The main difference with CoqOfOCaml is that this project focuses on the compilation of purely functional programs. There is a specific treatment of non-termination and partiality using the Bove-Capretta method [4].

Finally, the extraction mechanism [40] of Coq aims to compile Coq programs into OCaml programs. The challenges are not the same as with CoqOfOCaml: there are no effects to infer in Coq, but the type system is richer than the one of OCaml. In particular, dependent types and propositional values must be transformed in order to get an efficient evaluation of the compiled code. Contrary to CoqOfOCaml, the extraction mechanism of Coq is complete in the sense that is handles the full Coq language.

3.11 Conclusion

We have presented a compiler, CoqOfOCaml, which imports existing examples of OCaml programs to equivalent Coq programs. An effects system and a monadic translation is used to infer and represent the effects in Coq.

In the future we would like to investigate more the programming and proof techniques on Coq programs with effects. Among interesting problems are the extension of the effects system to handle parametrized effects, an implementation of an effects inference mechanism directly on Coq terms, the represen-

[18] The compiler hs-to-gallina was made by Gabe Dijkstra and is available on github.com/gdijkstra/hs-to-gallina.

tation of new effects (including concurrency), the design of reasoning rules on monadic programs with dependent types, and the certification of an extraction chain from Coq to OCaml with effects. We could also investigate a compilation of OCaml programs to the monad of Cybele, in order to import existing decision procedures written in OCaml to Coq.

Part III

Implementing asynchronous I/O in Coq

Chapter 4

Breakable Computations

The content of this chapter was presented at the WADT 2014 *workshop.*

4.1 Abstract

In the previous chapter, one of our challenges was to find a generic and composable encoding of programs with effects in the purely functional language Coq. Finding an expressive and usable encoding of effects is one of the keys to represent and reason about programs in Coq.

We introduce the notion of *breakable computations* which are programs with effects represented in Coq. We can pause and introspect a breakable computation at any point during its evaluation process. We can compose the effects in any order. This encoding is expressive enough to represent concurrent programs and various effects.

4.2 Introduction

We eventually run the effects of a program on a physical computer. The hardware of a computer enforces constraints over the kinds of effects which can be efficiently implemented. We will concentrate our efforts on the effects which crucially rely on the hardware for their implementation and which are complex to represent in a purely functional language. Indeed, we believe that, by restricting the list of the effects we focus on, we can find encodings of effects which are more usable and composable than the standard presentations of *monads* [63]. Among these effects are the mutable references, the exceptions, the non-termination, the inputs–outputs and the concurrency.

We used in CoqOfOCaml (page 6) the *sequential monads*, which are of type:

$$\mathcal{M}_{S,E}\, A = S \to (A + E) \times S$$

e_1		
$e_1\, s_1$	$=$	(e_2, s_2)
$e_2\, s_2$	$=$	(e_3, s_3)
$\vdots$		$\vdots$
$e_n\, s_n$	$=$	(r, s_{n+1})

Figure 4.1: Sequential execution.

with S a type of state and E a type of error. The monad $\mathcal{M}_{S,E}$ represents a state monad combined with an error monad. As we have seen, the monad $\mathcal{M}_{S,E}$ is composable and we can model any combination of state and error effects, as well as the non-termination and the inputs–outputs effects. However, we do not think that we can represent the concurrency effect with this monad.

To encode the concurrency effect, our idea is to enrich the type of our monad, so that we can express both the final result and the intermediate calculation steps of an effectful program. Indeed, once the intermediate calculation steps are made explicit, we can model the parallel evaluation of two programs by interleaving their calculation steps according to a scheduler. We represent an intermediate calculation step by a computation returning another computation representing the remaining calculation steps. Our first extended form of monad is the following:

$$\mathcal{M}_{S,E}\, A = S \to (A + E + \mathcal{M}_{S,E}\, A) \times S$$

We define this recursive type by induction in Coq. A computation takes a state of type S and either returns a final value of type A, an error of type E or another computation of type $\mathcal{M}_{S,E}\, A$, together with an updated state. We sequentially evaluate a computation e_1 on an initial state s_1 by repeatedly applying the resulting computations on the previous state (Figure 4.1). The final result is a value r of type A in case of success and of type E in case of error. The evaluation must terminate because e_1 was defined by induction.

To evaluate e_1 in parallel with another computation e_1', we can interleave the two computations (Figure 4.2). The structure of the computations allows us to evaluate them using different evaluation strategies without changing their definitions.

In order to make a clear distinction between the pure computations, which are not sensible to the evaluation order, and the effectful computations, we designed a second form of monad, the *breakable computations*[1] (a slight variation of the monad $\mathcal{M}_{S,E}$). In this chapter, we will introduce:

- the notion of breakable computations (page 75), programs with composable effects and explicit intermediate calculation steps, encoded in the purely functional language Coq;

[1] We provide a Coq definition of the breakable computations on github.com/clarus/coq-breakable-computations.

e_1			e_1'		
$e_1\, s_1$	$=$	(e_2, s_2)		$\bullet$	
	$\bullet$		$e_1'\, s_2$	$=$	(e_2', s_3)
$e_2\, s_3$	$=$	(e_3, s_4)		$\bullet$	
	$\bullet$		$e_2'\, s_4$	$=$	(e_3', s_5)
$\vdots$		$\vdots$	$\vdots$		$\vdots$

Figure 4.2: Concurrent execution.

- some examples of effects (page 79) represented using the breakable computations;
- some operators to model concurrent executions (page 81).

4.3 Definitions

In this section, we define the breakable computations and some basic operators.

4.3.1 Breakable computations

Definition 9 (Breakable computations)**.** *A* breakable computation *with a state type S, an error type E and a return type A is an element of the type $\mathcal{C}_{S,E}\, A$, inductively defined by:*

$$
\begin{array}{l}
\texttt{Inductive}\ \mathcal{C}_{S,E}\, A : \text{Type} := \\
\mid \text{Value} : A \to \mathcal{C}_{S,E}\, A \\
\mid \text{Error} : E \to \mathcal{C}_{S,E}\, A \\
\mid \text{Break} : (S \to \mathcal{C}_{S,E}\, A) \to (S \to S) \to \mathcal{C}_{S,E}\, A.
\end{array}
$$

A breakable computation is either a pure value of type A, an error of type E or a break. A break contains two functions depending on the current state, one to compute the next computation to execute and one to compute the next state. We pause the evaluation of a computation by stopping its evaluation at a break.

Definition 10 (Monadic operators)**.** *We define the two monadic operators* `ret`

and `bind` *as:*

$$\begin{array}{rcl} \text{ret} & : & A \to \mathcal{C}_{S,E}\, A \\ \text{bind} & : & \mathcal{C}_{S,E}\, A \to (A \to \mathcal{C}_{S,E}\, B) \to \mathcal{C}_{S,E}\, B \\ \text{ret}\, v & = & \text{Value}\, v \\ \text{bind}\, x\, f & = & \texttt{match}\ x\ \texttt{with} \\ & & \mid \text{Value}\, v \Rightarrow f\, v \\ & & \mid \text{Error}\, e \Rightarrow \text{Error}\, e \\ & & \mid \text{Break}\, x'\, s' \Rightarrow \text{Break}\, (\lambda s.\, \text{bind}\, (x'\, s)\, f)\, s' \end{array}$$

We define the *return* operator as expected. We define the *bind* operator by induction over x. If x is the pure value v, we apply the function f to v. If x is an error, we directly return this error. If x is a break, we return a break with the binding to f recursively applied on x'. The Coq checker accepts the recursive definition of `bind` as a terminating function since $x'\, s$ is a smaller term than x.

Definition 11 (Equality)**.** *We define the structural equality* $\sim$ *on the computations with the following rules:*

$$\textsc{Value}\ \frac{}{\text{Value}\, v \sim \text{Value}\, v} \qquad \textsc{Error}\ \frac{}{\text{Error}\, v \sim \text{Error}\, v}$$

$$\textsc{Break}\ \frac{\forall\, s,\ x_1\, s \sim x_2\, s \qquad \forall\, s,\ s_1\, s = s_2\, s}{\text{Break}\, x_1\, s \sim \text{Break}\, x_2\, s}$$

We verify that the structural equality $\sim$ is indeed a relation. Assuming the functional extensionality axiom[2], this relation is equivalent to the equality of Coq.

Theorem 2. *The breakable computations form a* monad *according to the structural equality* $\sim$*. They verify the following statements:*

$$\begin{cases} \text{bind}\, (\text{ret}\, x)\, f \sim f\, x \\ \text{bind}\, x\, \text{ret} \sim x \\ \text{bind}\, (\text{bind}\, x\, f)\, g) \sim \text{bind}\, x\, (\lambda x.\, \text{bind}\, (f\, x)\, g) \end{cases}$$

4.3.2 Sequential evaluation

The sequential evaluation is the simplest form of evaluation of the breakable computations. We evaluate a computation to a *sequential monad*, as defined page 61:

$$\mathcal{M}_{S,E}\, A = S \to (A + E) \times S$$

[2]The functional extensionality axiom is of type:

$$\begin{array}{l} \forall\, A\, B,\, \forall\, (f\, g : A \to B), \\ \quad (\forall\, x,\, f\, x = g\, x) \to f = g. \end{array}$$

with the following operations:

$$\begin{cases} \mathrm{ret}_{\mathcal{M}} : A \to \mathcal{M}_{S,E}\, A \\ \mathrm{bind}_{\mathcal{M}} : \mathcal{M}_{S,E}\, A \to (A \to \mathcal{M}_{S,E}\, B) \to \mathcal{M}_{S,E}\, B \\ e_1 \sim_{\mathcal{M}} e_2 \iff \forall\, s,\ e_1\, s = e_2\, s \end{cases}$$

Definition 12 (Sequential evaluation)**.** *We define the sequential evaluation of a computation by induction as follows:*

```
1  Fixpoint eval {S E A} (x : C.t S E A) : M.t S E A :=
2    fun s =>
3      match x with
4      | C.Value v => (inl v, s)
5      | C.Error e => (inr e, s)
6      | C.Break xs ss => eval (xs s) (ss s)
7      end.
```

We recursively apply the function `eval` while the computation x is a break, using the updated state of the break as a new state.

Theorem 3. *The sequential evaluation respects the following statements:*

$$\begin{cases} \mathrm{eval}\,(\mathrm{ret}\, v) \sim_{\mathcal{M}} \mathrm{ret}_{\mathcal{M}}\, v \\ \mathrm{eval}\,(\mathrm{bind}\, x\, f) \sim_{\mathcal{M}} \mathrm{bind}_{\mathcal{M}}\,(\mathrm{eval}\, x)\,(\lambda v.\, \mathrm{eval}\,(f\, v)) \\ x_1 \sim x_2 \to \mathrm{eval}\, x_1 \sim_{\mathcal{M}} \mathrm{eval}\, x_2 \end{cases}$$

4.3.3 Composition

Definition 13 (Composition)**.** *We define the composition of two sorts of computations by:*

$$\mathcal{C}_{S_1,E_1} \uplus \mathcal{C}_{S_2,E_2} = \mathcal{C}_{S_1 \times S_2, E_1 + E_2}$$

To compose two sorts of computations, we compose the state types and the error types. The composition of two states is the product of the states, the composition of two errors is the sum of errors. We introduce some operators to inject a computation into a larger type of computations.

Definition 14 (Lift)**.** *We define by induction the injections of computations with respect to the state type or with respect to the error type:*

$$\begin{cases} \mathrm{lift\_state} : \forall\,(S_1\, S_2\, E\, A),\ \mathcal{C}_{S_1,E}\, A \to \mathcal{C}_{S_1 \times S_2, E}\, A \\ \mathrm{lift\_error} : \forall\,(S\, E_1\, E_2\, A),\ \mathcal{C}_{S,E_1}\, A \to \mathcal{C}_{S,E_1+E_2}\, A \end{cases}$$

We omit to precise the definitions of the lifts since they are very straightforward. We check some basic properties to make sure that the lifts are well-defined.

Theorem 4. *The lifts operations commute with the monadic operators:*

$$\begin{cases} \text{lift\_state}\,(\text{ret}\,v) \sim \text{ret}\,v \\ \text{lift\_state}\,(\text{bind}\,x\,f) \sim \text{bind}\,(\text{lift\_state}\,x)\,(\lambda v.\,\text{lift\_state}\,(f\,v)) \end{cases}$$

$$\begin{cases} \text{lift\_error}\,(\text{ret}\,v) \sim \text{ret}\,v \\ \text{lift\_error}\,(\text{bind}\,x\,f) \sim \text{bind}\,(\text{lift\_error}\,x)\,(\lambda v.\,\text{lift\_error}\,(f\,v)) \end{cases}$$

Theorem 5. *The lifts operations commute together:*

$$\text{lift\_error}\,(\text{lift\_state}\,x) \sim \text{lift\_state}\,(\text{lift\_error}\,x)$$

We can simplify or reshape a state or an error type by applying an isomorphism. The following definitions will hold for the state type, but the same definitions would apply to the error type.

Definition 15 (Isomorphism)**.** *We define by induction the application of an isomorphism to the state of a computation:*

$$\begin{array}{l} \texttt{Fixpoint}\ \text{map\_state}\,(f : S_1 \to S_2)\,(g : S_2 \to S_1) \\ \quad (x : \mathcal{C}_{S_1,E}\,A) : \mathcal{C}_{S_2,E}\,A := \\ \quad \texttt{match}\ x\ \texttt{with} \\ \quad \mid \text{Value}\,v \Rightarrow \text{Value}\,v \\ \quad \mid \text{Error}\,e \Rightarrow \text{Error}\,e \\ \quad \mid \text{Break}\,x\,s \Rightarrow \text{Break}\,(\lambda s_2.\,\text{map\_state}\,f\,g\,(x\,(g\,s_2)))\,(\lambda s_2.\,f\,(s\,(g\,s_2))) \\ \quad \texttt{end}. \end{array}$$

The `map_state` operator verifies some basic properties.

Theorem 6. *The `map_state` function commutes with the monadic operators. Supposing two functions:*

$$f_{1\to2} : S_1 \to S_2 \quad ; \quad f_{2\to1} : S_2 \to S_1$$

we have:

$$\begin{cases} \text{map\_state}\,f_{1\to2}\,f_{2\to1}\,(\text{ret}\,v) \sim \text{ret}\,v \\ \text{map\_state}\,f_{1\to2}\,f_{2\to1}\,(\text{bind}\,x\,f) \sim \\ \quad \text{bind}\,(\text{map\_state}\,f_{1\to2}\,f_{2\to1}\,x)\,(\lambda v.\,\text{map\_state}\,f_{1\to2}\,f_{2\to1}\,(f\,v)) \end{cases}$$

Theorem 7. *The `map_state` function commutes with the functional operators. Supposing four functions:*

$$\begin{cases} f_{1\to2} : S_1 \to S_2 \quad ; \quad f_{2\to1} : S_2 \to S_1 \\ f_{2\to3} : S_2 \to S_3 \quad ; \quad f_{3\to2} : S_3 \to S_2 \end{cases}$$

we have:

$$\begin{cases} \text{map\_state}\,(\lambda s.\,s)\,(\lambda s.\,s)\,x \sim x \\ \text{map\_state}\,f_{2\to3}\,f_{3\to2}\,(\text{map\_state}\,f_{1\to2}\,f_{2\to1}\,x) \sim \\ \quad \text{map\_state}\,(f_{2\to3} \circ f_{1\to2})\,(f_{2\to1} \circ f_{3\to2})\,x \end{cases}$$

```
1  Definition raise {S E A} (e : E) : C.t S E A :=
2    C.Error e.
3
4  Fixpoint handle {S E1 E2 A} (x : C.t S (E1 + E2) A)
5    : C.t S E2 (A + E1) :=
6    match x with
7    | C.Value v => C.Value (inl v)
8    | C.Error (inl e1) => C.Value (inr e1)
9    | C.Error (inr e2) => C.Error e2
10   | C.Break xs ss => C.Break (fun s => handle (xs s)) ss
11   end.
```

Figure 4.3: Primitives for the error effect.

```
1  Definition read {S E : Type} : C.t S E S :=
2    C.Break (fun s => C.Value s) (fun s => s).
3
4  Definition write {S E : Type} (s : S) : C.t S E unit :=
5    C.Break (fun _ => C.Value tt) (fun _ => s).
```

Figure 4.4: Primitives for the state effect.

4.4 Examples

We present some examples of effects which can be encoded with the breakable computations.

4.4.1 Error

We implement the error effect using a non-empty error type. We define two functions, to raise and to handle the errors (Figure 4.3). These functions are parametrized by any state type S to be used in any context, but they do not access to the state value.

4.4.2 State

We implement the state effect with a non-void state type S and define two functions `read` and `write` to access the state (Figure 4.4). These functions need to use the constructor `Break` in order to control the state value, thus they will be sensible to the scheduling in a concurrent program.

4.4.3 Non-termination

As in CoqOfOCaml (page 53), we implement the non-termination effect using the technique of the *fuel*, combining an error and a state effect. The state represents the fuel, that is to say a integer decreasing at each iteration. If the fuel level reaches zero, then we return "`None`", else we return "`Some` r" with r the result.

```
Fixpoint f (x : A) (fuel : nat) : C.t S E B :=
  match fuel with
  | O => ret None
  | S fuel' => ... (f x' fuel') ...
  end.
```

We can abstract the usage of the fuel into a breakable computation with:

```
Definition use_fuel {S E A B : Type}
  (f : A -> nat -> C.t S E (option B))
  (x : A) : C.t (S * nat) (E + unit) B :=
  let! s_fuel : S * nat := State.read in
  let (s, fuel) := s_fuel in
  let! result := lift_error (lift_state (f x fuel)) in
  match result with
  | None => C.Error (inr tt)
  | Some y => ret y
  end.
```

We use the notation:

$$\texttt{let!}\ x := e_1\ \texttt{in}\ e_2$$

to represent the monadic bind:

$$\mathrm{bind}\, e_1\, (\lambda x.\, e_2)$$

4.4.4 Inputs–outputs

Given a type of inputs I, we implement a program with inputs as an input handler of type:

$$I \to \mathcal{C}_{S,E}\,\mathrm{unit}$$

To model the behavior of the program, we suppose that the list of inputs received by the program is known *a priori*. The function `loop_seq`:

$$\mathrm{loop\_seq} : (I \to \mathcal{C}_{S,E}\,\mathrm{unit}) \to \mathcal{C}_{S \times \mathrm{list}\, I,E}\,\mathrm{unit}$$

sequentially iterates the input handler over each input. We add the list of inputs to the state of the computation.

```
(* Sequentially apply a function f to each element of a list l. *)
Fixpoint iter_seq {S E A}
  (f : A -> C.t S E unit) (l : list A) : C.t S E unit :=
```

```
4     match l with
5     | [] => ret tt
6     | x :: l =>
7       let! _ := f x in
8       iter_seq f l
9     end.
10
11  (* Sequentially apply a handler to a list of events in the state.
12     Return a state with an empty list of events.
13  *)
14  Definition loop_seq {S E A}
15    (handler : A -> C.t S E unit) : C.t (S * list A) E unit :=
16    let! s_events : S * list A := State.read in
17    let (s, events) := s_events in
18    do! lift_state (List.iter_seq handler events) in
19    let! s_events : S * list A := State.read in
20    let (s, _) := s_events in
21    State.write (s, []).
```

For a type of outputs O, we represent the outputs made by the program by the ordered list of output events. The primitive `log`:

$$\texttt{Definition}\ \log\{E\,O\}\,(x : O) : \mathcal{C}_{\text{list}\,O,E}\,\text{unit} :=\\ \text{Break}\,(\lambda\_\,.\,\text{Value}\,\text{tt})(\lambda s.\,x :: s).$$

emits one output event and adds it to the current trace of outputs.

4.5 Concurrency

We model the concurrency with the concurrent operator `par`. The `par` operator (Figure 4.5) takes two computations and returns the couple of their results, evaluated concurrently according to a scheduling of type $\mathcal{H}$[3]:

$$\text{par} : \mathcal{C}_{S\times\mathcal{H},E}\,A \to \mathcal{C}_{S\times\mathcal{H},E}\,B \to \mathcal{C}_{S\times\mathcal{H},E}\,(A \times B)$$

A scheduling is an infinite list of booleans. We define the `par` operator by mutual induction on both of its arguments x and y. For each occurrence of a break in x and y, the `choose` function consumes a boolean to decide to execute either one step of x or one step of y. If one of the arguments if already a value, the `par` operator evaluates the other argument and returns the couple of results. If exactly one of the arguments is an error, this error is immediately returned. If both arguments are errors, the error to return is decided by consuming one boolean of the scheduling.

The result of a `par` being in the same kind of computations $\mathcal{C}_{S\times\mathcal{H},E}$ as its arguments, we can stack multiple applications of the `par` operator. In particular, we define the parallel iteration over a list by:

[3] We use the symbol $\mathcal{H}$ for the scheduling to represent a source of information, or entropy.

```
1  Definition choose {S E} : C.t (S * Entropy.t) E bool :=
2    let! state := State.read in
3    match state with
4    | (s, Streams.Cons b bs) =>
5      do! State.write (s, bs) in
6      ret b
7    end.
8
9  Fixpoint par {S E A B}
10   (x : C.t (S * Entropy.t) E A) (y : C.t (S * Entropy.t) E B)
11   : C.t (S * Entropy.t) E (A * B) :=
12   let fix par_aux y := ... in
13   match x with
14   | C.Value v_x => let! v_y := y in ret (v_x, v_y)
15   | C.Error e_x =>
16     match y with
17     | C.Value _ | C.Break _ _ => C.Error e_x
18     | C.Error e_y =>
19       let! b := choose in
20       if b then
21         C.Error e_x
22       else
23         C.Error e_y
24     end
25   | C.Break xs sxs =>
26     match y with
27     | C.Value v_y => let! v_x := x in ret (v_x, v_y)
28     | C.Error e_y => C.Error e_y
29     | C.Break ys sys =>
30       let! b := choose in
31       if b then
32         C.Break (fun s => par (xs s) y) sxs
33       else
34         C.Break (fun s => par_aux (ys s)) sys
35     end
36   end.
```

Figure 4.5: The `par` operator.

```
1  Fixpoint iter_par {S E A} (f : A -> C.t (S * Entropy.t) E unit)
2    (l : list A) : C.t (S * Entropy.t) E unit :=
3    match l with
4    | [] => ret tt
5    | x :: l =>
6      let! _ := par (f x) (iter_par f l) in
7      ret tt
8    end.
```

We define the `iter_par` function like the sequential iteration, except that we concurrently call `f` and `iter_par` with a `par` (line 6). We derive the function `loop_par` to concurrently run an input handler on an unordered list of input events:

$$\text{loop\_par} : (I \to \mathcal{C}_{S\times\mathcal{H},E}\,\text{unit}) \to \mathcal{C}_{(S\times\text{list}\,I)\times\mathcal{H},E}\,\text{unit}$$

We make a computation atomic by collapsing all its breaks into a single break:

```
1  Definition atomic {S E A} (x : C.t S E A) : C.t S E A :=
2    match x with
3    | C.Value _ | C.Error _ => x
4    | C.Break _ _ =>
5      C.Break
6        (fun s =>
7          match fst (eval x s) with
8          | inl v => C.Value v
9          | inr e => C.Error e
10         end)
11       (fun s => snd (eval x s))
12   end.
```

Unless if the argument x is already a pure value or an error, we evaluate x sequentially with the `eval` function (page 76). Thus, observationally, the computation must be executed in one step. The parallel evaluation of two atomic operations only depends on the choice of the first to execute.

4.6 Related work

THE IDEA OF REPRESENTING concurrent programs with effects in a purely functional language is not new.

In the functional languages Haskell [50] and Idris [7], concurrent programs are represented by a monad with axiomatized concurrent operators. A concurrent monad with no axioms [15] has also been proposed for Haskell. Unlike the breakable computations, this monad is expressed in a continuation passing style,

which may be harder to program with. In Coq, the project Ynot proposes an effectful monad to represent and reason about concurrent programs with a transactional heap [46].

4.7 Conclusion

We have presented the concept of *breakable monads*, which are programs with effects and explicit intermediate steps encoded in the purely functional language Coq. Thanks to these explicit steps, we defined an operator `par` to evaluate two computations in parallel. We have shown that various effect can be represented using the breakable computations, including the state, error, non-termination and inputs–outputs effects. These effects share the same presentation so that we can compose them naturally.

In the future, we would like to study variants of the computations or new proof techniques with the aim to formally specify and verify effectful programs encoded in Coq.

Chapter 5

Asynchronous Computations

The content of this chapter was presented on Hacker News[1].

5.1 Abstract

To communicate with the system and the users, programs must do inputs–outputs. But, for consistency reasons, we cannot directly express inputs–outputs in the purely functional languages used for theorem proving.

We introduce the notion of *asynchronous computations*, which represent effectful programs with asynchronous inputs–outputs in a dependently typed language. The computations are defined and specified without introducing new axioms. As an application we present Pluto, which we believe to be the first concurrent web server implemented in the Coq programming language.

5.2 Introduction

The use of a theorem prover as a programming platform is a promising technique, as it presents a unified platform to program, specify and formally verify an application. This is made possible thanks to the *Curry-Howard* correspondence [33] which states that, given the right presentation, a proof and a program are essentially the same thing. This presentation is the λ-calculus [14] and the *Type Theory*. The Coq system[2] provides one of the most mature implementation of Type Theory. However, for consistency reasons, purely functional languages such Coq are effects-free, whereas most realistic applications make effects such as inputs–outputs operations.

[1] The post is on https://news.ycombinator.com/item?id=8704804.

[2] The Coq system is available under LGPL license on coq.inria.fr.

In this chapter, we will introduce the framework CoqConcurrency[3], which extends in a consistent way the language of Coq to enable the programming of effectful and concurrent applications with inputs–outputs, using Coq as a programming language. We will present:

- the language of the *asynchronous computations*, a safe extension of the Coq language introducing uninterpreted constructors, to write concurrent applications with inputs–outputs, states and non-termination (page 87). The asynchronous inputs–outputs are modeled with an event system;
- a compilation chain and a runtime to execute the computations. The computations are compiled down to the effectful programming language OCaml and communicate with the operating system through a sanitized pipe. We formally specify the behavior of the computations (page 91);
- the web server Pluto[4], which is to our knowledge the first concurrent web server written in the programming language of Coq, implemented using our CoqConcurrency framework.

We can formalize and formally verify properties on the purely functional part of the computations, using the standard Coq techniques [59] for the formal certification of programs. However, the verification of the interactive parts of the computations outreaches the scope of this chapter.

5.3 General idea

We aim to implement a concurrent web server Pluto using Coq as a programming language. Coq being a purely functional language, our main challenge is to represent impure effects in the Coq system. We want to find a high-level presentation of effects which:

- preserves the consistency of the logic of Coq;
- can be executed with realistic efficiency;
- is convenient to program with.

In contrast with the previous chapter, we guide the design of the effect system by a runnable example, a web server, which we want to implement and be able to execute.

The main effect which we encode is the inputs–outputs effect. A realistic web server handling several clients simultaneously, we also encode the concurrency effect. Finally, we encode the non-termination effect since a web server should run indefinitely. In our implementation, we do not use the state and the error

[3] The CoqConcurrency framework is available under MIT license on github.com/coq-concurrency.

[4] Pluto is available under MIT license on github.com/coq-concurrency/pluto.

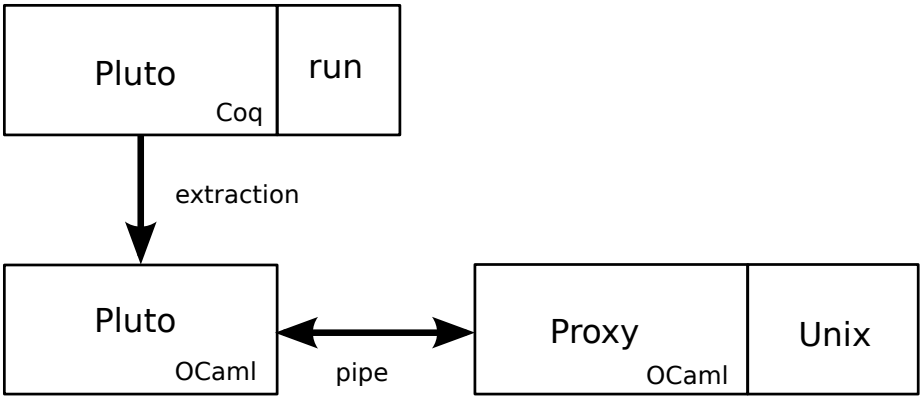

Figure 5.1: Runtime architecture.

effects. We do not need a state effect because we do not have a shared state between the client handlers nor mutable data structures. We use sum types instead of exceptions to represent errors. We choose to represent the concurrent inputs–outputs using an event system. Events can be emitted or received concurrently. Each request to the environment creates a fresh communication channel. A handler listen to the answers of the environment on each channel, and may close its channel if needed.

In Coq, we introduce the effectful primitives as constructors of a language of effectful *asynchronous computations*. Thus, we do not need any new axioms. This language of effectful computations is composable and includes the set of purely functional Coq programs.

We run these effectful primitives in two steps. First, we program in Coq a `run` function to evaluate the effectful computations of the Pluto web server, using some basic sequential OCaml primitives. Second, we program in OCaml an untrusted proxy, interfacing our inputs–outputs API with a Unix system. We connect both programs through a pipe transmitting events between the twos. This architecture is summarized on the Figure 5.1.

5.4 The language

To represent programs with concurrent and event-based inputs–outputs, we introduce the language of *asynchronous computations*. We define the computations in Coq by introducing new abstract combinators. The computational model is based on a soup of event handlers with a static shared memory.

5.4.1 Events

A program communicates with the outer world by creating some requests and by waiting for events answering to these requests. A request creates a communication channel with a fresh identifier, a kind of command and a request payload. The environment can answer to a request by zero, one or many events.

Definition 16 (Kind of commands)**.** *A* kind of commands *is an element of a type* `Command.t` *with a decidable equality and dependent types of request and answer payloads:*

$$\begin{cases} \text{eq\_dec} & : & \forall\,(c_1\,c_2 : \text{Command.t}),\ \text{option}\,(c_1 = c_2) \\ \text{request} & : & \text{Command.t} \to \text{Type} \\ \text{answer} & : & \text{Command.t} \to \text{Type} \end{cases}$$

We will use the decidable equality over the commands to implement the event dispatch.

Example For a program interacting with the terminal, we can define two kinds of commands:

```
Inductive Command.t :=
| Write
| Read.
```

with the following request and answer payloads:

```
Definition request c :=
  match c with
  | Write ⇒ string
  | Read ⇒ unit
  end.
```

```
Definition answer c :=
  match c with
  | Write ⇒ unit
  | Read ⇒ string
  end.
```

We can write or read a string to the terminal. To both commands the system is supposed to answer exactly once, but we do not provide a way to specify the number of answers.

5.4.2 Memory

The memory is a typed list of cells shared by all the event handlers. This memory can be updated by atomic read and write operations. We could add other kinds of operations, such as locking or transactions.

Definition 17 (Signature)**.** *A* signature σ *is a list of types.*

Definition 18 (Memory)**.** *A* memory *of signature* σ *is an element of the type* $\mathcal{M}\,\sigma$*, inductively defined by:*

```
Inductive M :=
| Nil : M []
| Cons : ∀ A σ, A → M σ → M (A :: σ).
```

Each cell of the memory is typed by an element of its signature. To manipulate this memory conveniently, we introduce a class of references.

Definition 19 (Reference)**.** *A* reference *of type A in a memory of signature σ is an instance of the class $\mathcal{R}\,A\,\sigma$, defined by:*

$$\begin{array}{l}\texttt{Class}\ \mathcal{R}\,A\,\sigma : \text{Type} := \text{New}\,\{\\ \quad \text{read} : \mathcal{M}\,\sigma \to A;\\ \quad \text{write} : \mathcal{M}\,\sigma \to A \to \mathcal{M}\,\sigma;\\ \}.\end{array}$$

We provide instances of the reference class so that, for a memory of signature σ and a type of cell A appearing just once in σ, Coq automatically infers the unique instance of $\mathcal{R}\,A\,\sigma$.

Example We can access a memory of signature $\sigma = [\mathbb{N}; \text{bool}]$ with two references, which are the unique instances of the classes $\mathcal{R}\,\sigma\,\mathbb{N}$ and $\mathcal{R}\,\sigma\,\text{bool}$. A possible memory state is the couple of the two following cells:

$$\{12; \text{true}\}$$

5.4.3 Computations

The computations represent Coq programs interacting with the environment. The computations are composed of non-interactive parts written in the purely functional language of Coq and of interactive parts using special abstract operators.

Definition 20 (Asynchronous computation)**.** *An* asynchronous computation *of signature σ and return type A is an element of the type $\mathcal{C}\,\sigma\,A$, inductively defined by:*

$$\begin{array}{l}\texttt{Inductive}\ \mathcal{C}\,\sigma : \text{Type} \to \text{Type} :=\\ \mid \text{Ret} : \forall\, A,\ A \to \mathcal{C}\,\sigma\,A\\ \mid \text{Bind} : \forall\, A\,B,\ \mathcal{C}\,\sigma\,A \to (A \to \mathcal{C}\,\sigma\,B) \to \mathcal{C}\,\sigma\,B\\ \mid \text{Read} : \forall\, A, \grave{}\{\mathcal{R}\,A\,\sigma\} \to \mathcal{C}\,\sigma\,A\\ \mid \text{Write} : \forall\, A, \grave{}\{\mathcal{R}\,A\,\sigma\} \to A \to \mathcal{C}\,\sigma\,\text{unit}\\ \mid \text{Send} : \forall\, A\,(c : \text{Command.t}),\ \text{Command.request}\,c \to\\ \quad A \to (A \to \text{Command.answer}\,c \to \mathcal{C}\,\sigma\,(\text{option}\,A)) \to \mathcal{C}\,\sigma\,\text{unit}\\ \mid \text{Exit} : \forall\, A,\ \mathcal{C}\,\sigma\,A.\end{array}$$

An asynchronous computation can be either:

- $\text{Ret}\,e$, the purely functional Coq expression e;
- $\text{Bind}\,e_1\,e_2$, the sequencing of the computation e_1 with the computation e_2 applied to the result of e_1;
- $\text{Read}\,r$, the reading of the reference r;

- Write $r\,v$, the writing of the reference r with the value v;
- Send $c\,r\,s\,h$, the sending of a new request of kind c with a payload r. To handle the answers of the environment, we start a handler h with an initial state s. The handler h takes two arguments, the current state of the handler and an answer from the environment. The handler returns a new state to continue to listen for the answers, or the special value `None` to stop listening. The `Send` operator is *asynchronous* and returns immediately while the handler is listening for the answers;
- Exit, a special instruction which halts the program and stops all the event handlers.

The handler state is local to each handler. It is important for an event handler to stop once all the expected event have been received, in order to free the table of active handlers. We will use the following notations to sequence two computations:

$$\begin{cases} \texttt{let!}\ x := e_1\ \texttt{in}\ e_2 & := & \text{Bind}\, e_1\, (\texttt{fun}\ x \Rightarrow e_2) \\ \texttt{do!}\ e_1\ \texttt{in}\ e_2 & := & \text{Bind}\, e_1\, (\texttt{fun}\ \_ \Rightarrow e_2) \end{cases}$$

Example Using the commands defined page 88, here is a "Hello world" program:

```
Definition hello_world : C [] unit :=
  Send Command.Write "Hello world!" tt
    (fun _ _ ⇒ Exit).
```

We do not need a shared memory, thus we use an empty signature `[]`. We emit the request `Command.Write` to the environment with the payload "Hello world!". Since we run the handler just once, we do not need a handler state and choose the state `tt` (the `unit` value). The payload of the answer to a write is also of type `unit`, so both arguments of the handler can be replaced by an underscore. In the body of the handler, we run the `Exit` instruction to terminate the program.

This program is quite involving for a "Hello world". We can replace it with a simpler:

```
Definition hello_world : C [] unit :=
  write "Hello world!" Exit.
```

assuming a function `write` parametrized by a continuation k:

```
Definition write (s : string) (k : C [] unit) : C [] unit :=
  Send Command.Write s tt (fun _ _ ⇒ k).
```

Due to the type of the operator `Send`, most of the computations we write are written in a continuation passing style. We could relate the use of continuations for web programming to the work of Christian Queinnec [52].

$$\begin{cases} \text{seq} : \forall\, A,\, (\text{unit} \to \text{unit}) \to \\ \quad (\text{unit} \to A) \to A \\ \text{launch} : \text{string} \to \text{pipe} \\ \text{print\_line} : \text{string} \to \text{pipe} \to \text{unit} \\ \text{fold\_lines} : \forall\, A,\, \text{pipe} \to A \to \\ \quad (A \to \text{string} \to \text{option}\, A) \to \text{unit} \end{cases}$$

Figure 5.2: The main impure Coq primitives.

5.5 Implementation

We implement the runtime of the language of the computations into two distinct parts. The first part aims to compile the computations down to OCaml and is mostly written in Coq with basic sequential OCaml primitives. The second part is a proxy which aims to communicate with the operating system and is fully written in OCaml. The two parts are connected through a Unix pipe in such a way that we only need to trust the compilation part.

5.5.1 Compilation

To compile a Coq computation, we execute its effects by applying the evaluation function `run` of type:

$$\text{run} : \forall\, \sigma,\, \mathcal{M}\, \sigma \to (\text{list string} \to \mathcal{C}\, \sigma\, \text{unit}) \to \text{unit}$$

and compile the resulting program down to OCaml using the extraction mechanism of Coq [40]. The function `run` takes as arguments a signature σ, an initial memory of signature σ, a computation parametrized by a list of strings (the command line arguments) and returns the pure value `unit`. Since we cannot directly execute the effects of a computation in Coq, the function `run` calls some axiomatized Coq primitives which are extracted to their equivalent in OCaml. Among these primitives are the axioms given on the Figure 5.2. The primitive `seq` sequences the evaluation of two (impure) expressions. The primitive `launch` launches a process and returns a pipe to communicate with it. The primitive `print_line` sends a message in a pipe. The primitive `fold_lines` listens to a pipe by recursively applying a function f with a state of type A on each message. If the function f returns a new state, the function f is applied again on the next message with the new state. If the function f returns `None`, the function `fold_lines` terminates.

A pseudo-code of a simplified function `run` is given on the Figure 5.3. For brevity, we do not present the full function `run` whose definition also accounts for the manipulation of the shared memory, the exit instruction and the errors cases. The complete implementation is given in the `coq:concurrency:system`[5]

[5] The `coq:concurrency:system` package is available under MIT license on github.com/coq-concurrency/system.

```
run e :=
  pipe := launch "proxy";
  heap := new_heap ();
  eval e;
  for each (id, answer) do
    (handler, s) := heap[id];
    s' := eval (handler s answer);
    if s' = ∅ then
      heap[id] := ∅;
    else
      heap[id] := (handler, s');
  end
```

Figure 5.3: Pseudo-code of a simplified function `run`.

package.

The `run` function is single-threaded and interleaves the executions of the handlers as the answers arrive. First, we launch the proxy communicating with the operating system using the primitive `launch`. Second, we initialize a heap of handlers. A heap of handlers is a set of handlers associated to a unique communication channel identifier and a local state. Third, we execute the computation. Whenever an instruction `Send` is encountered, we create a new channel identifier, allocates a new handler and send the request to the proxy. Finally, we listen to the messages of the proxy using the primitive `fold_lines`. Each message is a couple of a channel identifier and an answer payload. We evaluate the handler associated to the channel identifier and get its new local state s'. If the new state is empty, we disable the handler. Otherwise, we update the local state of the handler.

5.5.2 Proxy

The proxy listens to the requests of the handlers, executes them and sends back the answers of the system. The commands implemented by the proxy are listed on the Figure 5.4. We implement the proxy in OCaml using the lightweight threads library Lwt [62]. We use this library as a communication layer with the system, but not as an implementation of our scheduler defined by the function `run` (Figure 5.3). The command `Log` prints a message on the terminal, the command `FileRead` gets the content of a file and the command `Time` gets the current time in seconds since the Unix epoch. The remaining commands are used to manipulate a TCP socket from the server side. The commands `ServerBind` and `ClientRead` may answer several times as the clients get connected or send messages.

$$\begin{cases} \text{Log} : \text{string} \to \text{bool} \\ \text{FileRead} : \text{string} \to \text{option string} \\ \text{Time} : \text{unit} \to \mathbb{N} \\ \text{ServerBind} : \mathbb{N} \to \text{option id} \\ \text{ClientRead} : \text{id} \to \text{option string} \\ \text{ClientWrite} : \text{id} \times \text{string} \to \text{bool} \\ \text{ClientClose} : \text{id} \to \text{bool} \end{cases}$$

Figure 5.4: Commands implemented by the proxy.

5.5.3 Protocol

The compiled computations and the proxy exchange serialized messages through a pipe. A request from a computation is a text line of the form:

$$command\_id\_payload$$

with *id* a fresh channel identifier (an integer) and *payload* the request payload. We encode the string payloads in base64 using an OCaml library[6]. Likewise, the answers from the proxy are of the form:

$$command\_id\_payload$$

5.5.4 Specification

We do not specify the proxy or the operating system, because we think it would be unrealistic to expect the system not to have bugs. Instead, we isolate the computations from the system using the pipe mechanism. By doing so, we reduce the interactions of the computations with the system to the reading or the printing of text lines in a pipe.

We specify the compilation of the computations by defining in Coq a pure version of the `run` function with no axioms:

$$\text{pure\_run} : \forall\, \sigma,\, \mathcal{M}\, \sigma \to \mathcal{C}\, \sigma\, \text{unit} \to \text{list input} \to \text{list output}$$

This function defines what are the expected output messages sent to the proxy, for the execution of a computation with a list of input messages sent by the proxy. More precisely, we expect that, for each signature σ, each initial memory m, each computation e, each list of command line arguments a and each list of input messages l, if the execution of the OCaml extraction of:

$$\text{run}\, \sigma\, m\, e$$

with the command line arguments a got the inputs l from the proxy, then the OCaml extraction sent the list of messages:

$$\text{run\_pure}\, \sigma\, m\, (e\, a)\, l$$

[6] We use the base64 package available on the OPAM repository for OCaml.

to the proxy. The formal verification of this specification would outreach the scope of this chapter[7].

5.6 The Pluto web server

As a challenge, we used the computations to implement the Pluto[8], which is (to our knowledge) the first concurrent web server implemented in the Coq programming language. Pluto serves static websites using the HTTP protocol version 1.1. In our experience, Pluto is efficient enough to serve personal websites.

5.6.1 Overview of the code

The Pluto web server first binds a TCP server socket to wait for the clients. We loop on the incoming clients:

```
1  ServerSocket.bind port_number (fun client =>
2    match client with
3    | None => Log.write "Server socket failed." (fun _ => C.Exit)
4    | Some client => handle_client website_dir client
5    end)))
```

and run the `handle_client` function on each new client. Here is a slightly simplified version of the `handle_client` function:

```
1  Definition handle_client (website_dir : LString.t)
2    (client : ClientSocketId.t) : C.t [] unit :=
3    ClientSocket.read client "" (fun read request =>
4    match request with
5    | None => ... (* We close the socket. *)
6    | Some line =>
7      let read := read ^ line in
8      match Request.parse read with
9      | inr err => C.Ret (Some read) (* We wait for more data. *)
10     | inl (Request.New (Get, url, protocol) headers) =>
11       do!
12         let url := website_dir ^ url in
13         match Url.parse url with
14         | inr err => ... (* Wrong url. *)
15         | inl (Url.New file_name _) =>
16           Time.get (fun time =>
17           let time := Moment.of_epoch time in
```

[7] In particular, this would require to formally verify the extraction mechanism of Coq.

[8] Pluto is available under MIT license on github.com/coq-concurrency/pluto web server. Pluto is also a dwarf planet of the solar system. At the time of the design of this web server, the space probe New Horizons was on its way to give us the first close-by pictures of Pluto.

```
18          File.read file_name (fun result =>
19          let answer := ... in
20          answer_client client url answer))
21        end in
22      (* We continue to listen, clearing the data. *)
23      C.Ret @@ Some []
24    end
25  end).
```

We loop on the client messages (line 3), with a local state concatenating all the client messages (initially, the empty string ""), since an HTTP request may contain several lines. If the message is valid (line 6), we parse the client request. If the request is incomplete (line 9), we wait for more client messages. Otherwise (line 10), we parse the URL. If the URL is valid (line 15), we get the current time (line 16), read the requested file (line 18) and generates an answer (lines 19 to 24). Depending on the result of the read operation, the answer may be the content of the web page or a 404 error. Finally, we send the answer to the client by calling the `answer_client` function (line 20). While the current file is being served, we concurrently listen for new requests (line 23).

5.6.2 Dependencies

While implementing the web server Pluto, we realized that some programming libraries for Coq were missing. We created the library ListString[9], which contains many functions to handle strings of characters. These functions were useful for the parsing and the pretty-printing of HTTP messages. We also created the library Moment[10] to manipulate dates and times. In particular, Moment can convert dates expressed in seconds since the Unix epoch to dates expressed in the Gregorian calendar and pretty-print them in the RFC 1123 format.

5.7 Related work

The use of purely functional programming languages to write programs with inputs–outputs has already been extensively studied.

The project Ynot [47] introduces an axiomatized monad in Coq to write programs with imperative effects, like pointer manipulations. The programs can be certified and formally verified using Hoare logic reasonings [32]. An extension to the Ynot monad enables the representation of programs with inputs–outputs, such as web applications [42]. However, contrary to our CoqConcurrency library, the Ynot monad cannot represent concurrent programs and asynchronous inputs–outputs.

[9] The library ListString is available under MIT license on github.com/clarus/coq-list-string.
[10] The library Moment is available under MIT license on github.com/clarus/coq-moment.

The language Idris [7] is a purely functional programming language with primitives to enable effectful programming. Idris provides a monad for synchronous inputs–outputs [6] together with an effect system [8] based on the algebraic effects. We can compile Idris programs to the C language or the Erlang language, the concurrency aspects being handled by an actors and messages system.

The Reflex platform [54] introduces a domain specific language to represent the orchestration part of reactive systems and automatically verify some formal properties. In particular, the authors implemented a web server using the Reflex platform. In contrast to our CoqConcurrency library, Reflex uses Coq as a proof language but not as a programming language. Indeed, the Reflex expressions are represented in Coq through a *deep embedding*.

The Bedrock system provides the Bedrock IL, a low-level assembly language deeply embedded in Coq, together with definitions and automation procedures to formally specify and verify Bedrock programs. A multi-threaded web application had been implemented and verified using Bedrock [12]. Like the Relfex system, the Bedrock system uses Coq as a proof language but not as a programming language.

Finally, many mostly-purely functional languages such as Haskell (strictly speaking, Haskell is not a purely functional language because programs may not terminate and pattern-matchings may not be complete) implement inputs–outputs and concurrency though impure monads [49]. However, the type system of Haskell is not strict and expressive enough to permit theorem proving.

5.8 Conclusion

We introduced the language of the *asynchronous computations* to represent concurrent programs with inputs–outputs in the purely functional language Coq. We compile the computations down to the OCaml language and connect them to the operating system through a sanitized pipe. As an example, we implemented the first concurrent web server Pluto using Coq as a programming language.

In the future, we would like to simplify the language of computations to a direct-style language, in order to avoid the use a continuation passing style. We would also like to design specification and verification techniques to formally certify the interactive parts of the programs written using the computations.

Part IV

Developing with asynchronous I/O in Coq

Chapter 6

Interactive Computations

The work of this chapter was published at the FormaliSE 2015 *workshop and on Hacker News*[1].

6.1 Abstract

INTERACTIVE PROGRAMS, like user interfaces, are hard to formally specify and thus to prove correct. Some ideas coming from the functional programming languages have been successful to improve the way we write safer programs, but these ideas mostly apply to code fragments with no inputs–outputs.

Using the purely functional language Coq, we present a new technique to represent interactive programs, or *interactive computations*, and formally verify properties expressed as *use cases*. To this end we introduce the notion of *runs*, well-typed schema of interactions between an environment and a program. We design and certify a blog system as an illustration. Our approach generalizes unit-testing techniques and outlines a new method for mechanically assisted checking of effectful functional programs.

6.2 Introduction

PROVING CORRECT interactive programs is challenging. Indeed, interactive programs are hard to reason about because they communicate with an outer environment (the operating system, the network, the user, ...) which may be under-specified and non deterministic. In the previous chapter, we have seen how to implement interactive programs with asynchronous inputs–outputs in Coq. We will now investigate how to formally verify interactive programs.

[1] The post is on https://news.ycombinator.com/item?id=9037115

To start with, we restrict ourselves to sequential interactive programs with synchronous inputs–outputs.

In software development processes, we often employ *use cases* [34] to specify and verify programs with inputs–outputs. A use case is basically a sequence of interaction steps between a program and an environment, in order to denote some functional properties. We can verify a use case by manually running the program it specifies and by looking at its trace, or by using automated unit-testing. However, unit-testing methods are never complete as soon as there is an infinite set of possible values for the inputs of the program.

We developed a new method to write and certify interactive programs by a formal use cases analysis in Coq. We illustrate our method by programming and certifying the interactive blog system ChickBlog[2]. In this chapter, we introduce and present:

- the notion of *interactive computations* (page 102) to express interactive computations with synchronous inputs–outputs in the purely functional language Coq;
- a semantic for the computations as the set of the well-typed *runs*. A run is a program which describes both the trace, which can be built interactively, and the result of a computation (page 103);
- a new technique to express and verify properties expressed as use cases over interactive programs (page 108). Use cases are represented through parametrized runs;
- a symbolic debugger for interactive programs (page 109), to assist the design of use cases and to spot bugs. This debugger relies on the existing tactic mode [21] of Coq, using this mode to explore the execution paths of an interactive computation;
- the blog system ChickBlog (page 100), implemented and certified using our method. The user can login, add, edit or remove a post though a web interface. This blog system is compiled to an executable version using as an intermediate language the OCaml programming language.

6.3 A challenge

Our challenge is to both develop and certify the blog system ChickBlog in Coq. This challenge is interesting because a blog is a realistic example of a program with interactive user interactions.

Using this blog system, the user is able to login, add, edit or delete a post. We save the posts on the file system, with one file per post. We consider a very

[2]The blog system ChickBlog is available under MIT license on github.com/clarus/coq-chick-blog.

simple login system (without passwords) to concentrate on the architecture. The user interacts with the blog through HTTP requests.

We use an implementation[3] of the HTTP protocol in the OCaml language, a general purpose programming language. The blog itself is entirely written in Coq. We formally express and prove some properties by reasoning by use cases. To generate a runnable executable of the blog, we compile the Coq code into an OCaml code and then use the OCaml compiler. Our trust base is composed of the Coq system, the extraction process from Coq to OCaml, the OCaml compiler and runtime, and the implementation of HTTP in OCaml.

6.4 Computations and runs

We will introduce the notion of *interactive computations* and the notion of *runs*, in order to represent and give a semantics to interactive programs in Coq. These interactive programs will communicate with the system by calling some special commands.

6.4.1 Commands

A command is the value emitted during a call of a program to the system. Let `Command.t` be the type of commands and:

$$\text{answer} : \text{Command.t} \to \text{Type}$$

the dependent type of the answers to these commands.

Example To read the content of a file and to log messages on the user terminal, we can use a type of commands with two constructors:

$$\begin{cases} \text{ReadFile} & : & \text{string} \to \text{Command.t} \\ \text{Log} & : & \text{string} \to \text{Command.t} \end{cases}$$

and the following types of answers:

$$\begin{cases} \text{answer ReadFile} & = & \text{option string} \\ \text{answer Log} & = & \text{unit} \end{cases}$$

Both kinds of commands are parametrized by a string, respectively a filename and a user message. To the `ReadFile` commands, the system may answer the string representing the file content, or `None` in case of error. The `Log` commands always return the unit value.

[3] We use the `cohttp` library, available on the OPAM repository for OCaml.

```
1  Definition print_readme : C.t unit :=
2     Call (ReadFile "README") (fun text =>
3     match text with
4     | None => Ret tt
5     | Some text =>
6       Call (Log text) (fun _ =>
7       Ret tt)
8     end).
```

Figure 6.1: The `print_readme` procedure.

6.4.2 Interactive computations

Definition 21 (Interactive computation)**.** *The* interactive computations *returning a value of type A are represented by the type $\mathcal{C}\,A$, inductively defined in Coq by*[4]*:*

$$\begin{aligned}&\texttt{Inductive}\ \mathcal{C}\ (A : \text{Type}) : \text{Type} :=\\ &|\ \texttt{Ret} : \forall\,(x : A),\ \mathcal{C}\,A\\ &|\ \texttt{Call} : \forall\,(c : \text{Command.t}),\ (\text{answer}\,c \to \mathcal{C}\,A) \to \mathcal{C}\,A.\end{aligned}$$

An interactive computation can be either:

- a pure expression x of type A;
- a call to the environment with an argument c of type `Command.t` and a *handler* waiting for an answer of type `answer` c, dependent on the value of the command.

The role of a computation is to combine pure code fragments and to sequence calls to the system, in order to form arbitrarily complex programs interacting with the environment.

Example On the Figure 6.1 we present a computation printing the content of a file on the user terminal. We call an external procedure to read the file `README` on line 2. The variable `text` is set to the answer of the call, which is expected to be the content of the `README` file. If the content is `None` (in case of error), we return the unit value on line 4. If the content is some text, we print it on the standard output using the command `Log` on line 6. Once the printing function has terminated, we return the unit value.

We only assumed there will be one answer of the right type for each call. We will now see how to specify the values of these answers.

[4] We define the computations in the source file `src/Computation.v` which is available on github.com/clarus/coq-chick-blog.

6.4.3 Runs

To reason about interactive programs we need to give a semantics to the computations. The semantic of a computation is defined by the type of all its runs. A *run* is a program executing a computation by providing an explicit answers to each call.

Definition 22 (Run)**.** *The type of the* runs $\mathcal{R}$ *is a type parametrized by a type* A *and a computation* c *of type* $\mathcal{C}\,A$*, defined inductively by*[5]*:*

$$
\begin{aligned}
&\texttt{Inductive}\ \mathcal{R}\ (A : \text{Type}) : \mathcal{C}\,A \to \text{Type} :=\\
&|\ \texttt{RunRet} : \forall\,(x : A),\ \mathcal{R}\,A\,(\texttt{Ret}\,x)\\
&|\ \texttt{RunCall} : \forall\,(c : \text{Command.t})\,(a : \text{answer}\,c),\\
&\quad \forall\,\{\mathit{handler} : \text{answer}\,c \to \mathcal{C}\,A\},\ (\mathcal{R}\,A\,(\mathit{handler}\,a)) \to\\
&\quad \mathcal{R}\,A\,(\texttt{Call}\ c\ \mathit{handler}).
\end{aligned}
$$

A run can be either:

- a run of a `Ret` that carries the pure value x returned by a computation;
- a run of a `Call` of a command c that received an answer a of the corresponding type and a run of a handler applied to the answer a.

We do not explicitly write the *handler* terms since they are already in the definition of the computation and thus can be automatically inferred by Coq (the implicit parameters are declared into braces). A run describes both an execution trace of a computation and the result of its evaluation with this trace. Thus, we can extract the result of a run with the function eval:

$$
\begin{aligned}
&\texttt{Fixpoint}\ \text{eval}\,\{A : \text{Type}\}\,\{c : \mathcal{C}\,A\}\,(r : \mathcal{R}\,A\,c) : A :=\\
&\quad \texttt{match}\ r\ \texttt{with}\\
&\quad |\ \texttt{RunRet}\ x \Rightarrow x\\
&\quad |\ \texttt{RunCall}\ c\ a\ h\ r \Rightarrow \text{eval}\,r\\
&\quad \texttt{end}.
\end{aligned}
$$

We define `eval` by induction over a run. We recurse until we find the inner `Ret` and return its value. The `eval` function returns a value of type A as expected. Similarly, we extract the trace of a run:

$$
\begin{aligned}
&\texttt{Fixpoint}\ \text{trace}\,\{A : \text{Type}\}\,\{c : \mathcal{C}\,A\}\,(r : \mathcal{R}\,A\,c)\\
&\quad : \text{list}\,\{c' : \text{Command.t}\ \&\ \text{answer}\,c'\} :=\\
&\quad \texttt{match}\ r\ \texttt{with}\\
&\quad |\ \texttt{RunRet}\ x \Rightarrow [\,]\\
&\quad |\ \texttt{RunCall}\ c'\ a\ h\ r \Rightarrow (c', a) :: \text{trace}\,r\\
&\quad \texttt{end}.
\end{aligned}
$$

A trace is a list[6] of dependent couples of commands and answers of the corresponding type. We recurse over a run, accumulating the command and the answer of each call in a list.

[5] We define the runs in the file `src/Spec.v` available on github.com/clarus/coq-chick-blog.

[6] Since Coq is a normalizing language, by construction, the trace is always a finite list and the evaluation of a run always terminates.

```
1  Definition run_print_readme : R unit print_readme :=
2    RunCall (ReadFile "README") (Some "Blabla") (
3    RunCall (Log "Blabla") tt (
4    RunRet tt)).
```

Figure 6.2: Example of run of the `print_readme` program.

Example For the `print_readme` program on the Figure 6.1, we give an example of run on the Figure 6.2. This run tells us that the program `print_readme` will call the command `ReadFile "README"`. If we answer `Some "Blabla"`, then it will call the command `Log "Blabla"`, to which the only possible answer is `tt`. Then the program terminates without any other calls. We can see this run as a form of specification by the example of the program `print_readme`, describing its behavior only when we answer `Some "Blabla"` to the `ReadFile` operation.

We have given a formal definition and a formal semantic of the interactive computations in Coq. We will now see how we used this notion of computations to build a blog server.

6.5 Programming the blog

We will present the code of the blog system ChickBlog[7], and explain how we used the notion of computations to implement the inputs–outputs operations.

6.5.1 The server handler

The main function of the blog server is of type[8]:

$$\text{server} : \text{Path.t} \to \text{Cookies.t} \to \mathcal{C}\,\text{Response.t}$$

This function handles one request from the client. A request is a path (an URL, like `/login`) and the status of the client's cookies. A response is a MIME type, a new set of cookies and a body (typically some HTML content). The `server` function is an *interactive computation* since it does calls to the system. The state of the blog is saved on the file system and accessed through system calls. In total, the blog system itself is made of 786 lines of Coq code. The `server` function is pure, expect for the uninterpreted inputs–outputs operations. In particular, the `server` function is deterministic, cannot return any exceptions and always terminates. This property is given to us for free thanks to the strict type system of Coq.

[7] The blog system ChickBlog is available under MIT license on github.com/clarus/coq-chick-blog.

[8] We define the `server` function in the source file `src/Main.v` which is available on the repository github.com/clarus/coq-chick-blog.

Constructor	Argument	Root path
`NotFound`		
`WrongArguments`		
`Static`	list string	`/static`
`Index`		`/`
`Login`		`/login`
`Logout`		`/logout`
`PostAdd`		`/posts/add`
`PostDoAdd`	string × date	`/posts/do_add`
`PostEdit`	string	`/posts/edit`
`PostDoEdit`	string × string	`/posts/do_edit`
`PostDoDelete`	string	`/posts/do_delete`
`PostShow`	string	`/posts/show`

Table 6.1: Constructors of the `Path.t` type.

The path of a request, initially a string URL, is parsed in Coq to the sum type `Path.t`[9]. A sum type is a union of different types, each introduced by a constructor. The constructors of the `Path.t` type are given in the Table 6.1. This explicit sum type also describes the web API of the blog application. The `NotFound` and `WrongArguments` constructors are for ill-formed requests. `Static` retrieves static content such as CSS files. `Index` shows the main page. We use `Login` and `Logout` to login and logout (there are no passwords or user names). `PostAdd` shows the form to add a post, `PostDoAdd` effectively add a post. So do `PostEdit` and `PostDoEdit` to edit a post. `PostDoDelete` removes a post. Finally, `PostShow` shows the content of a post.

6.5.2 Edit a post

We present the implementation of the handler `post_do_edit`[10] of `PostDoEdit` requests (Figure 6.3). This handler generates a response for a request of the form:

$$/\texttt{posts/do\_edit}/\mathit{url}?\texttt{content} = \mathit{content}$$

We check if the user is logged (line 3). If so, we call the `header` function of the module `Helpers` to get the meta-data of a post from the file system (line 7). Its type is:

$$\forall\, A,\ \text{string} \to (\text{option}\,\text{Post.Header.t} \to \mathcal{C}\ A) \to \mathcal{C}\ A$$

This function `Helpers.header` waits for a continuation, a function returning the next computation. The notation `let!` (line 7) is a syntactic sugar to call a

[9] We define the type `Path.t` in the source file `src/Request.v` which is available on the repository github.com/clarus/coq-chick-blog.

[10] We define the `post_do_edit` function in the source file `src/Main.v` which is available on the repository github.com/clarus/coq-chick-blog.

```
1  Definition post_do_edit (is_logged : bool)
2    (url : string) (content : string) : C Response.t :=
3    if negb is_logged then
4      ret Response.Forbidden
5    else
6      let! is_success : bool := fun k ⇒
7        let! header := Helpers.header url in
8        match header with
9        | None ⇒ k false
10       | Some header ⇒
11         let file_name := posts_directory ++
12           Post.Header.file_name header in
13         call! is_success :=
14           UpdateFile file_name content in
15         k is_success
16       end in
17     ret (Response.PostDoEdit url is_success).
```

Figure 6.3: Code of the `post_do_edit` handler.

function with a continuation:

$$\texttt{let!}\ x := e_1\ \texttt{in}\ e_2 \quad \Longleftrightarrow \quad e_1\,(\lambda x.\, e_2)$$

We program by continuations to compose computations. Since the *monad* structure can generalize the continuation passing style [45], we could have added the monadic bind operator:

$$\texttt{Bind} : \forall\, A\, B,\ \mathcal{C}\, A \to (A \to \mathcal{C}\, B) \to \mathcal{C}\, B$$

but we preferred to keep our number of primitives as small as possible, in the hope to simplify the reasoning over the runs[11].

If the header is not available, we return `false` for the success status to the continuation k (line 9). Otherwise we call the command `UpdateFile` to update the content of the post on the file system (line 14). We give the list of calls used by the blog[12] on the Table 6.2. The notation `call!` (line 13) is a syntactic sugar for a call to the environment:

$$\texttt{call!}\ x := c\ \texttt{in}\ e \quad \Longleftrightarrow \quad \texttt{Call}\ c\ (\lambda x.\, e)$$

Finally, we return a page `PostDoEdit` with a link to the original post and the success status of the update (line 17). The `Response.t` type is a sum type,

[11] See for example the definition of an invariant over the cases of a computation page 111.

[12] We define the calls used by the blog in the source file `src/Computation.v` available on github.com/clarus/coq-chick-blog.

Command	Argument	Answer
`ReadFile`	string	option string
`UpdateFile`	string × string	bool
`DeleteFile`	string	bool
`ListPosts`	string	option (list header)
`Log`	string	unit

Table 6.2: Calls used by the blog.

with one constructor per kind of page. A purely functional pretty-printer[13] then renders the corresponding HTML content.

We have seen how to program a blog system in a type-safe manner in Coq using the notion of interactive computations. In the next section, we will see how to compile this blog system.

6.6 Compiling the blog

We will explain how we compile the interactive computation of the blog system to an executable program, using an automatic translation to the OCaml language[14].

The Coq programming language is a purely functional language. We can represent the inputs–outputs operations but we cannot execute them. To do so, we use the *extraction* mechanism [40], which compiles Coq programs to the impure programming language OCaml. We introduce some uninterpreted constants and extract them to specific impure expressions in OCaml. For example:

```
1 Parameter printl : String.t -> t unit.
2 Extract Constant printl => "Lwt_io.printl".
```

declares the Coq constant `printl` and associates it to the OCaml function:

```
Lwt_io.printl
```

which prints a line on the standard output. After extraction, this function will effectively display a line on the screen. As mentioned earlier, this compilation process is part of the trusted base. We link the extracted code to the OCaml library CoHTTP to handle the HTTP protocol. CoHTTP creates the main program loop, waiting for HTTP requests. The event-based concurrency model is managed by the Lwt library [62], a cooperative lightweight threads library for OCaml.

[13] We do not verify the pretty-printer.

[14] We define the compilation of the computations in the file `src/Extraction.v` available on github.com/clarus/coq-chick-blog.

1. we add a new post as an authenticated user;
2. we edit this post with some text s;
3. we show this post and check that it contains the text s we edited.

Figure 6.4: Use case of the creation of a new blog post.

6.7 Specifying the blog

THE TYPE SYSTEM OF COQ already provides some safety properties for the computations, like the termination. We will go further by formally verifying *use cases* and invariants.

6.7.1 Use cases

A use case is a form of specification of an interactive program, expressed as a scenario of interaction between the program and its environment. We give an example of an informal use case on the Figure 6.4. This use case describes the creation of a new blog post. The user represents the environment of the blog and successively creates, edits and displays a blog post. We will formalize the notion of use cases in the setting of the computations.

Definition 23 (Use case)**.** *A* use case *over a computation c of type $\mathcal{C}\,A$ is a run r parametrized by some type P:*

$$\begin{cases} P : \text{Type} \\ r : P \to \mathcal{R}\,A\,c \end{cases}$$

For the use case of the Figure 6.4, a parameter can be a couple of a post title t and of a post content s. A run $r\,(t, s)$ will answer to the calls of the blog system in order to simulate the actions of the user described in the use case. We say that a use case is valid if the use case is well-typed.

To be relevant, the expression of a use case should be as clear as possible. Indeed, while we can formally verify that a use case is correct by typing, we cannot verify that a use case corresponds to the original intent of a programmer unless we provide a simple enough expression. The use cases being expressed as parametrized runs, which are programs answering to the calls of a computation, we hope that our specifications by use cases will provide a clear and familiar formalism for the programmers.

In the followings, to keep the explanations short, we will study the simpler use case of the index page service (Figure 6.5).

1. we connect to the index page URL;
2. the blog calls the file system to list the available posts;
   - in case of error, a log message is printed on the server console;
3. the index page is displayed with the list of posts.

Figure 6.5: Use case for the index page.

```
1  Definition index_ok (cookies : Cookies.t)
2    (headers : list Header.t)
3    : Run.t (Main.server Path.Index cookies).
4    simpl.
7    apply (RunCall (ListPosts _) (Some headers)).
8    apply (RunRet (Response.Index
9      (Cookies.is_logged cookies)
10     headers)).
11 Defined.
```

Figure 6.6: Formal use case for the index page.

6.7.2 Symbolic debugger

To write the runs of the use cases, we leverage the tactic mode of Coq. We normally use the tactic mode to prove theorems reasoning step by step. By using this mode to specify the computations, we get some form of interactive debugger for computations: it is possible to evaluate a computation stepping through each call, like we would iterate into the reasoning steps of a theorem. At each call, we must provide an explicit answer to go to the next call. The trace of the user interactions with the debugger is then a *run* of a computation. Crucially, this debugger is symbolic: Coq being designed to manipulate symbolic expressions, we do not need to instantiate variables with concrete values. So the interactions with the debugger can easily describe a parametrized run, that is to say a use case.

We give a formalization of the use case `index_ok`[15] for the index page (Figure 6.5) on the Figure 6.6. This use case describes the execution of the server handler by a run parametrized with a cookie and a list of post headers. We do not need to execute these runs: because `index_ok` is well-typed, we know that these runs are all correct[16]. These runs interpret one call of kind `ListPosts` and display the list of posts.

To construct this use case using the debugger (the tactic mode), we enter

[15] We define the `index_ok` use case in the file `src/Spec.v` available on github.com/clarus/coq-chick-blog.

[16] In contrast, to verify this use case by testing would require to execute the runs for each parameter, what is impossible here since the set of parameters is infinite.

```
1  Definition index (is_logged : bool)
2    : C.t Response.t :=
3    call! posts := ListPosts posts_dir in
4    match posts with
5    | None =>
6      do_call! Log ("Cannot open the " ++
7        posts_dir ++ " directory.") in
8      ret (Response.Index is_logged [])
9    | Some posts =>
10     ret (Response.Index is_logged posts)
11   end.
```

Figure 6.7: Source code of the index function.

the type of `index_ok` (the three first lines). The Coq interpreter replies:

```
1  1 subgoals
2  cookies : Cookies.t
3  headers : list Header.t
4  ______________________________________(1/1)
5  Run.t (Main.server Path.Index cookies)
```

This means that we have two symbolic parameters, `cookies` and `headers`, and aim to construct a run of the server handler applied to the index path and the cookies. We enter the `simpl` command on line 4 to partially evaluate the computation:

```
Main.server Path.Index cookies
```

The Coq engine uses the fact that `Path.Index` is a concrete value to simplify the application of `Main.server` to the application of the `Main.Controller.index` handler:

```
Main.Controller.index (Cookies.is_logged cookies)
```

We get:

```
1  1 subgoals
2  cookies : Cookies.t
3  headers : list Header.t
4  ______________________________________(1/1)
5  Run.t (Main.Controller.index
6    (Cookies.is_logged cookies))
```

We can guess what will be the next call of this computation by unfolding the definition of `Main.Controller.index` with the command `unfold` or by directly looking at its source code on line 3 on Figure 6.7. We guess that is a `ListPosts`

command applied to some folder:

$$\texttt{call! posts} := \texttt{ListPosts posts\_dir in} \ldots$$

We answer to this command the list of post headers `Some headers` on line 7:

$$\texttt{apply (RunCall (ListPosts \_) (Some headers))}.$$

The Coq system unifies modulo evaluation the computation:

$$\texttt{Main.Controller.index (Cookies.is\_logged cookies)}$$

with a computation of the form:

$$\texttt{Call (ListPosts } \alpha\texttt{) (fun } a \Rightarrow \beta\texttt{)}$$

It infers that α equals `posts_dir` and that β equals the lines 4 to 11 of the `index` function, that is to say, once `posts` is replaced by the answer `Some headers`:

$$\texttt{ret (Response.Index (Cookies.is\_logged cookies) headers)}$$

Thus, Coq answers:

```
1 subgoals
cookies : Cookies.t
headers : list Header.t
______________________________________(1/1)
Run.t (C.Ret (Response.Index
  (Cookies.is_logged cookies)
  headers))
```

Since we are on a `Ret` expression, the evaluation is terminated and we can conclude by the line 8 on the Figure 6.6, which explicitly states the expected result. In particular, we require the response to be the index page and to include the list of `headers`.

Likewise, we have defined a use case for the index page when there are errors from the file system, by answering the value `None` to the call `ListPosts`. We have also verified the add, edit and show use case of the Figure 6.4. In our experience, defining a use case is similar to writing a unit-test. Indeed, a unit-test for an interactive program is basically a trace of all the responses of the environment to the program. This is exactly the same for a use case, with the possibility to have symbolic answers, representing an infinite set of possible values.

6.7.3 Invariants

We now verify some invariants about the blog system. For example, we prove than an unauthenticated user cannot make a request which generates calls modifying the file system. To do so, we define a predicate:

$$\text{is\_read} : \text{Command.t} \to \text{bool}$$

```
Inductive read_only : C A → Prop :=
| RoRet : ∀ (x : A), read_only (Ret x)
| RoCall :
  ∀ c (h : answer c → C A),
  is_read c = true →
  (∀ (a : answer c), read_only (h a)) →
  read_only (Call c h).
```

Figure 6.8: A computation free of write operations.

to check that a command does not modify the file system. The function `is_read` answers `false` for the kinds of commands which may modify the file system, such as `UpdateFile`. By induction over the cases of a computation, we define what is a computation free of write operations (Figure 6.8). A computation is free of write operations if, for any answers to any of its calls, the computation does not make a call which does not respect the `is_read` predicate. We exploit the fact that a computation is defined by only two cases to give a simple definition of our `is_read` predicate.

To prove our claim, we reason by disjunction on the URL paths to we show that the `read_only` predicate is valid for any computation handling a request from an unauthenticated user.

Theorem 8 (Unharmful unauthenticated users)**.** *An unauthenticated user cannot modify the content of the file system of the server*[17]*:*

$$\forall\,(p : \text{Path.t}),\ \text{read\_only}\ (\text{Main.server}\ p\ \text{Cookies.LoggedOut})$$

6.8 Related work

Effectful functional programming have been intensively studied by the community of the Haskell language, and large realistic programs have been written using this functional language. One of the ideas popularized by Haskell is the IO monad to express impure computations and to verify by typing the isolation of pure and impure expressions. Simon Peyton Jones wrote a nice paper [49] on the IO monad and related techniques. We wanted to build upon this experience exploring new solutions, thanks to the more powerful type system of Coq featuring dependent-types and propositional types.

The Ynot project [47] studied the use of a parametrized monad to represent and reason about impure computations in Coq. This project focused more on the imperative and low-level memory management, using Hoare-logic [32] with pre- and post-conditions together with separation logic [53]. They explored

[17] We verify this property in the file `src/Spec.v` available on github.com/clarus/coq-chick-blog.

extensions to reason about inputs–outputs and have written and mechanically certified a web application [42]. Unlike our work, their application is specified by an invariant over the execution trace of the program. They do not study the verification of properties expressed as use cases.

The algebraic effects and handlers [51], a generic framework to represent effects in a compositional way in purely functional languages, led to a lot of research about proven safe effectful programs. This framework is more powerful than ours because it can represent many different kinds of effects, like non-determinism, exceptions, or states, and can combine them in a generic way. This power has a cost: the algebraic effects are also more complex to define and understand. It could be interesting to see if our notion of *interactive computations* can be viewed as a particular case of algebraic effect.

The dependently typed programming language Idris [7] proposes an implementation of algebraic effects [8]. Edwin Brady and Simon Fowler show how to specify the rules of a game or a web protocol and how to verify by typing their implementations in Idris [9, 24]. These work mostly focus on expressing invariants and building the right primitives to write correct-by-construction programs. By contrast, we focused on specifications by use cases and on a tool, the symbolic debugger, to express these cases interactively.

Ur/Web [13] is a functional programming language and a platform for the web development made by Adam Chlipala. We can cite the BazQux Reader[18], a commercial RSS feed reader, as a successful application of the Ur/Web platform. The language Ur does not feature full dependent types, but can generate formally valid and unexploitable web pages and SQL requests thanks to its rich type system and its integrated platform. Combining our formalism of use cases with the ideas of the Ur/Web platform could be an interesting subject.

6.9 Conclusion

We have presented the notions of *interactive computations* and *runs* to represent and to give a semantic to interactive programs in the purely functional language Coq. We have shown how to verify invariants on a computation using predicates inductively defined on the combinators of the computations. We have introduced a new technique to formalize and verify properties expressed as use cases. The tactic mode of Coq provides a symbolic debugger to write and verify these use cases interactively. To illustrate this approach, we developed and certified a blog system in Coq.

In the following chapters, we will extend our techniques to a wider class of programs. In particular, we will study concurrent programs with asynchronous calls, expressed in a monad with a join primitive to launch concurrent operations.

[18] https://bazqux.com/

Chapter 7

Concurrent Computations

7.1 Abstract

The use cases are a specification method used in software engineering to specify interactive programs. Some argue that the use cases have the advantage of being simple to understand, by describing scenarios of interactions between a program and its environment. However, they are usually written in informal languages and often verified just by program testing.

We present a formalization of the use cases in the dependently typed programming language Coq. We apply the use cases to formally verify programs written in the same language Coq, extended with typed primitives for concurrency and inputs–outputs. To simplify the design of the use cases, we exploit the proof mode of Coq as a symbolic debugger to interactively write correct-by-construction use cases from a program definition. We present several composition methods in order to specify programs in the large.

7.2 Introduction

The specification by use cases is a method used to write program specifications. Basically, we define a use case as a scenario of interactions between a program and its environment (in a given context), and a scenario as a list of inputs–outputs operations between the program and its environment.

To illustrate this technique, we consider the example of a distant shell with authentication (Figure 7.1). The shell asks the login and the password of the user (lines 2-3), and asks it again while the password is incorrect (lines 4-7). Finally, the shell runs and prints the outputs of the commands typed by the user (lines 8-11). We specify this shell with several use cases between the program and three *actors* representing the environment: the user, the authentication process and the system running the commands.

```
1  shell() :=
2    login = get_login()
3    password = get_password()
4    until is_valid(login, password)
5      login = get_login()
6      password = get_password()
7    done
8    while c = ask_command()
9      output = run(c)
10     print(output)
11   done
```

Figure 7.1: Pseudo-code of a shell with authentication.

1. the user enters the login *login*;
2. the user enters the password *password*;
3. the program asks for the validity of the couple *login* and *password* to the authentication server and gets the answer `true`;
4. the user requests a command execution.

Figure 7.2: Example of use case for the shell.

We give an example of use case with no password errors on the Figure 7.2. We parametrize this use case by two strings *login* and *password*. On step 4, we *include* another use case specifying how a user executes the commands. We can also *extend* this use case by describing how to react to special events, like for example to a terminal error when asking for the password.

As for any program specifications, we must ask two questions:

1. Does this specification represent the true intents of the specifier?
2. Does our implementation respect this specification?

We cannot answer to the first question in a certain way, because the true intents of a specifier are an intuitive human idea. However, we should design our specification language so that it is clear and readable, with common requirements expressed in a simple way. We classify the program specifications into two categories, according to the point of view taken by the specifier:

- the *internal specifications*, where the specifier explicits some invariants which the program will ensure in all acceptable execution contexts. The environment is just supposed to verify some pre-conditions useful to prove the invariants and the attention of the specifier is focused on the program itself;

- the *external specifications*, where the specifier describes the behavior of the program in the valid execution environments. The focus is drawn on the definition of a set of acceptable execution environments, together with the outputs which the program must respond to the inputs of the environment.

The distinction between these two categories occurs especially in the case of interactive programs with inputs–outputs. The internal specifications tend to be useful to describe the algorithmic parts of a program, whereas the external specifications tend to be useful to describe the interactive parts. These two kinds of specifications are complementary. The use cases are a form of external specification and aim to be simple to understand by giving concrete execution scenarios, even if it can be hard to convince ourself that the use cases cover all the kinds of scenarios we wanted to specify.

To the second question of the correctness of the implementation according to the specification, we can answer with a high degree of certainty by using formal methods. To this end, we continue the preliminary work of the chapter on the *interactive computations* (page 99) to present a formal definition of the use cases in type theory, together with a method to write and verify these use cases. All our work is formalized in Coq and usable to write runnable and verified application[1]. In this chapter we will present:

- the notion of *concurrent computations* (page 119), which are potentially non-terminating concurrent programs with inputs–outputs, written in the dependently typed language Coq. We give a formal semantic of the concurrent computations (page 123);
- a formalization of the use case specifications in Coq (page 126). To our knowledge, this is the first definition of the use cases specifications in type theory;
- an alternative representation of the use cases, using a symbolic execution of the computations in a debugger mode to guide both the definition and the verification of the use cases (page 128). In this representation, a use case is defined as a dual and valid-by-construction programs modeling the environment of a computation;
- a study of some composition and programming patterns in order to specify programs in the large, by adapting UML use case diagrams relations and by presenting new techniques (page 132 and page 137).

7.3 General idea

To verify that the use case of the Figure 7.2 is valid, we can start by writing a test program instrumenting the shell in order to play the use

[1] Our developments are available on the project website coq.io.

```
1  test_shell():=
2    login = random_string()
3    password = random_string()
4    enter(login)
5    enter(password)
6    request = get_auth_request()
7    assert request is
8      "check(login, password)"
9    answer_auth(true)
10   test_commands()
```

Figure 7.3: Pseudo-code of a test for the use case.

```
1  test_shell_coq(login, password):=
3    enter(login)
4    enter(password)
5    request = get_auth_request()
6    assert request is
7      "check(login, password)"
8    answer_auth(true)
9    test_commands_coq()
```

Figure 7.4: Pseudo-code of a test written in Coq.

case (Figure 7.3). This pseudo-code simulates a user entering a login and a password (lines 2-5), verifies that the shell asks for the authentication permission (lines 7-8), generates a fake authentication permission (line 9) and test the commands loop by calling `test_commands` (line 10). Although very useful, this test can only check the use case for a finite set of *login* and *password* values. Thus, we may miss some corner case errors (for example, what should happen if the login is empty?), or miss some execution paths.

Tests as specifications To formally verify the use case, our first idea is to write the exact same test in the Coq programming language, relying on the fact that "well-typed programs do not go wrong" to make sure that this test will succeed for each instance of *login* and *password* (Figure 7.4). When looking at the pseudo-code, the only thing which changed is that the randomly generated variables are now explicit parameters of the test. Thus, if this test "type checks", we know that for each instance of *login* and *password* the test will succeed[2]. This test provides both a *formalization* and a *validation* of our use case. Since we represent use cases as tests, we hope this formalism to be simple to grasp and easy to integrate in a standard programming workflow.

[2]We assume that we are in a total language, that is to say that each well-typed term reduces to a value. Exceptions and non-termination are forbidden.

Formally verifying use cases by writing tests in Coq does not come for free. Because Coq is a purely functional language, we need to introduce new operators to add inputs–outputs commands, non-termination or concurrency to the language (page 119). We also need to add constructs to write test programs which instrument other Coq programs (page 128). Despite the inference mechanisms of the Coq system, the type-checking of the test programs cannot always be automatic. It is in fact undecidable in general, because we could embed arbitrarily complex assertions into a use case.

A prover as a debugger Our second idea is to use the prover mode of Coq as a symbolic debugger to both write and verify the use cases (page 129). Indeed, the Coq system being a theorem prover based on type theory, in which the notion of evaluation is crucial, Coq is equipped with various symbolic evaluation and unification algorithms [65]. These algorithms proved to be useful to check our use cases. As an example, the verification of the use case for the shell is done automatically by symbolically executing the test interacting with the shell (the only non-automatic action is the unfolding of the login loop, due to technicalities with the co-induction).

By using the Coq proof mode (a mode to construct a term piece by piece), the symbolic execution of the test appears step by step, helping the specifier to understand more clearly the use case. For more complex use cases, where automatic symbolic execution can fail, we manually apply lemma to simplify the expressions which get stuck. For example, with the hypothesis $y \geq 0$ the expression:

$$\texttt{if } x^2 + y \geq 0 \texttt{ then } A \texttt{ else } B$$

could manually by reduced to A, by applying the right mathematical lemma. The ability to use both automatic and manual reduction rules offers a lot of control to write both simple and complex use cases.

7.4 Concurrent computations

To formalize concurrents programs with inputs–outputs in type theory, we introduce the notion of *concurrent computations*. The concurrent computations are a combination of purely functional expressions with concurrency operations and special requests to the environment.

7.4.1 Effects

An *effect* is a set of commands. The commands represent calls to the environment, for example to do inputs–outputs operations, to raise exceptions or to modify state variables.

```
Inductive command : Type:=
| Print (s : string)
| ReadLine.

Definition answer (c : command)
  : Type:=
  match c with
  | Print _ ⇒ unit
  | ReadLine ⇒ option string
  end.

Definition terminal_effect : Effect:=
  Effect.New command answer.
```

Figure 7.5: Declaration of an effect to interact with the terminal.

Definition 24 (Effect declaration)**.** *An* effect declaration *is an element of type* `Effect`*, defined by*[3]*:*

```
Record Effect := New {
  command : Type;
  answer : command → Type }.
```

This record contains two types:

- the `command` type, the type of the parameter of a call;
- the `answer` type, dependent on the parameter of a call, the type of answers for a call.

An effect declaration is purely abstract and does not specify the semantic of the calls, only their types. We will see how to define an effect later (page 137). The separation of the definitions from the declarations permits to provide many implementations for each effect declaration, either by simulation using purely functional constructs or by compilation to more primitive effects.

The Figure 7.5 is an example of declaration of effect, with two operations to display or read a message on the terminal. A command is either a `Print` parametrized by a message string or a `ReadLine` with no parameters. The environment's answer to a `Print` command is of type `unit`, the singleton type. The environment's answer to a `ReadLine` command is of type `option string`: the answer is either `None` in case of error or `Some` s (with s a string) in case of success. In this example, the command is a sum type representing two different kinds of operations. We can generalize this construction to compose any two declarations of effects together.

[3] We define the effect declarations in the file `src/Effect.v` available on github.com/coq-io/io.

Definition 25 (Effect composition)**.** *We define the composition of two effect declarations E_1 and E_2 by*[4]*:*

```
Definition compose (E1 E2 : Effect) : Effect :=
  Effect.New
    (Effect.command E1 + Effect.command E2)
    (fun c ⇒
      match c with
      | inl c1 ⇒ Effect.answer E1 c1
      | inr c2 ⇒ Effect.answer E2 c2
      end).
```

We define the type of commands of a composition of two effects E_1 and E_2 as the sum type of the commands of E_1 and E_2. The type of answers is defined accordingly.

7.4.2 Computations

The Coq programming language is purely functional, in the sense that each expression is terminating, deterministic and cannot raise exceptions. To represent concurrent programs with effects, which we name the *concurrent computations*, we introduce a set of unevaluated constructors. One advantage of writing the computations directly in Coq is that we will not verify a model but the actual implementation of our programs, modulo the trusted compilation process. We can also exploit the usual guarantees given by the type checker or the usual techniques [59] to verify the purely functional parts of the computations.

Definition 26 (Concurrent computations)**.** *The* concurrent computations *returning a value of type A using the effect declared by E are the values of the type $\mathcal{C}\,E\,A$, defined by co-induction as follows*[5]*:*

```
CoInductive C (E : Effect) : Type → Type :=
| Ret : ∀ A, ∀ (e : A), C E A
| Call : ∀ c, C E (Effect.answer E c)
| Let : ∀ A B, ∀ (e1 : C E A) (e2 : A → C E B), C E B
| Join : ∀ A B, ∀ (e1 : C E A) (e2 : C E B), C E (A × B).
```

A concurrent computation can be either:

- `Ret` e, the pure expression e of type A. This expression can be any valid Coq expression of type A;
- `Call` c, the call of a command c. The return value of a call depends on the value of c and is given by Effect.answer $E\,c$;

[4] We define the effect composition in the file `src/Effect.v` available on github.com/coq-io/io.

[5] We define the concurrent computations in the file `src/C.v` available on github.com/coq-io/io.

- `Let` $e_1\, e_2$, the application of the function e_2 (returning a computation of type $\mathcal{C}\, E\, B$) to the computation e_1 returning a value of type A. This application is sequential: the value v_1 of the computation e_1 is computed before the evaluation of $e_2\, v_1$;
- `Join` $e_1\, e_2$, the parallel evaluation of the computations e_1 and e_2. This operator waits till the evaluation of both e_1 and e_2 is completed and returns the couple of results of e_1 and e_2. The interleaving of the evaluation of e_1 and e_2 is unspecified.

Because the computations are co-inductively defined, they can describe either terminating or non-terminating programs. The idea of computations is inspired by the concept of *monads* [63], where impure expressions are isolated from the pure world and sequenced by two operators `Return` and `Bind`. These operators correspond to the `Ret` and `Let` primitives of the computations. However, we prefer not to call a computation a monad because the monadic equations such as:

$$\forall\, x\, f,\ \texttt{Bind}\,(\texttt{Return}\, x)\, f = f\, x$$

are not verified by the computations[6].

7.4.3 Evaluation

The inputs–outputs operations of the computations cannot be run inside Coq. To execute the effects of a computation, we compile the computations to the programming language OCaml using a customized version of the extraction mechanism [40] of Coq. We implement the commands and the `Join` operator using the OCaml library Lwt [62] (a library for asynchronous inputs–outputs). This compilation chain is not proven correct yet and is part of our trust base. The compilation infrastructure does not introduce new axioms to the Coq formalization of the computations.

7.4.4 Illustation

We will use the notations:

$$\begin{cases} \texttt{let!}\ x := e_1\ \texttt{in}\ e_2 \\ \texttt{do!}\ e_1\ \texttt{in}\ e_2 \end{cases}$$

for, respectively:

$$\begin{cases} \texttt{Let}\ e_1\ (\texttt{fun}\ x \Rightarrow e_2) \\ \texttt{Let}\ e_1\ (\texttt{fun}\ \_ \Rightarrow e_2) \end{cases}$$

As an illustration, we define the computation `your_name` on the Figure 7.6. The type of `your_name` is "$\mathcal{C}$ terminal_effect unit". This means that this is a computation returning the value of the singleton type `unit` and doing some effects to interact with the terminal. We print a welcome message on the terminal (line 2)

[6] At least using the default equality of Coq.

```
1  Definition your_name : C terminal_effect unit :=
2    do! Call (Print "What is your name?") in
3    let! name := Call ReadLine in
4    match name with
5    | None ⇒ Ret ()
6    | Some name ⇒ Call (Print ("Hello " + name))
7    end.
```

Figure 7.6: The `your_name` computation.

and ask for the name of the user (line 3). In case of error of the `ReadLine` command, we return the unit value (line 5). In case of success (line 6), we print the user name prefixed by the "Hello" message.

7.5 Semantics of the computations

THE DEFINITION OF THE COMPUTATIONS introduces operators to call effectful commands or to define concurrent programs. These operators are declared but not defined, in order to make the framework of the computations as generic as possible. The semantics of the computations will keep the definition of these operators abstract. We will define two equivalent kinds of semantics[7], the semantic of the *traces* and the semantic of the *runs*.

7.5.1 Traces

We first define the semantic of a computation as the set of its acceptable traces of interactions with the environment.

Definition 27 (Traces)**.** *For an effect declaration E, the type of* traces $\mathcal{T}\,E$ *is the type co-inductively defined by*[8]*:*

$$\begin{array}{l}\texttt{CoInductive}\ \mathcal{T}\,(E : \text{Effect}) : \text{Type}\ :=\\ |\ \texttt{TRet} : \mathcal{T}\,E\\ |\ \texttt{TCall} : \forall\, c\,(a : \text{Effect.answer}\,E\,c),\ \mathcal{T}\,E\\ |\ \texttt{TLet} : \mathcal{T}\,E \to \mathcal{T}\,E \to \mathcal{T}\,E\\ |\ \texttt{TJoin} : \mathcal{T}\,E \to \mathcal{T}\,E \to \mathcal{T}\,E.\end{array}$$

A trace is either:

- the empty trace `TRet` for a purely functional computation;

[7] We will consider as equal two expressions considered as equal by Coq. In particular, any expression can be assimilated to its (unique) normal form according to the reductions rules of Coq. We do not enforce any reduction strategies.

[8] We define the traces in the file `src/Trace.v` available on github.com/coq-io/io.

- the trace of a call $\texttt{TCall}\, c\, a$ for a call of a command c with the answer a;
- the sequential composition $\texttt{TLet}\, t_1\, t_2$ of two traces;
- the concurrent composition $\texttt{TJoin}\, t_1\, t_2$ of two traces.

The traces are possibly infinite trees of interaction events with two kinds of nodes, `TLet` and `TJoin`, to represent sequentiality or concurrency. We associate to each computation a set of acceptable traces.

Definition 28 (Valid traces)**.** *For a computation e of type $\mathcal{C}\, E\, A$, a trace t of type $\mathcal{T}\, E$ and a value v of type A, the predicate $e \xrightarrow{t} v$ asserts that the computation e accepts the trace t and returns the value v, by using the answers to the calls from the trace. This predicate is co-inductively defined by the rules*[9]*:*

$$\frac{}{\texttt{Ret}\, v \xrightarrow{\texttt{TRet}} v} \qquad \frac{}{\texttt{Call}\, c \xrightarrow{\texttt{TCall c a}} a} \qquad \frac{e_1 \xrightarrow{t_1} v_1 \qquad e_2\, v_1 \xrightarrow{t_2} v_2}{\texttt{Let}\, e_1\, e_2 \xrightarrow{\texttt{TLet t}_1\, \texttt{t}_2} v_2}$$

$$\frac{e_1 \xrightarrow{t_1} v_1 \qquad e_2 \xrightarrow{t_2} v_2}{\texttt{Join}\, e_1\, e_2 \xrightarrow{\texttt{TJoin t}_1\, \texttt{t}_2} (v_1, v_2)}$$

This predicate both asserts which traces are accepted by a program and which value is returned by a computation given an accepted trace. By construction, this predicate is confluent[10]: for a computation e and an accepted trace t, there is a unique value v such that $e \xrightarrow{t} v$. The environment can answer any value of the right type to a call from a computation. If one has to precise the semantic of the commands, this semantic can be given later in the form of a specification over the computations, for example. The scheduling of the operations for the join of two computations cannot be specified using the traces semantic.

7.5.2 Runs

As an alternative, we define the semantic of a computation as the type of all its possible runs. A run contains three informations: a trace, the validity of this trace and a return value.

Definition 29 (Run)**.** *For a computation e of type $\mathcal{C}\, E\, A$, the type of its* runs

[9] We define the valid traces in the file `src/Trace.v` available on github.com/coq-io/io.

[10] Because of the absence of induction-recursion in Coq, we could not explicit this confluence by mutually defining the type of the accepted traces and an evaluation function from the computations with valid traces to the values. However, this formalization may be possible in alternative theorem provers like Agda.

giving a result v of type A is the type $\mathcal{R}\,e\,v$, co-inductively defined by:

$$\begin{array}{l}
\texttt{CoInductive}\ \mathcal{R}\,(E : \text{Effect}) : \forall\, A,\, \mathcal{C}\,E\,A \to A \to \text{Type}\ :=\\
\mid\ \texttt{RRet} : \forall\, A\,(v : A),\, \mathcal{R}\,(\texttt{Ret}\,v)\,v\\
\mid\ \texttt{RCall} : \forall\, c\,(a : \text{Effect.answer}\,E\,c),\, \mathcal{R}\,(\texttt{Call}\,c)\,a\\
\mid\ \texttt{RLet} : \forall\, A\,B\,(e_1 : \mathcal{C}\,E\,A)\,v_1\,(e_2 : A \to \mathcal{C}\,E\,B)\,v_2,\\
\quad \mathcal{R}\,e_1\,v_1 \to \mathcal{R}\,(e_2\,v_1)\,v_2 \to \mathcal{R}\,(\texttt{Let}\,e_1\,e_2)\,v_2\\
\mid\ \texttt{RJoin} : \forall\, A\,B\,(e_1 : \mathcal{C}\,E\,A)\,v_1\,(e_2 : \mathcal{C}\,E\,B)\,v_2,\\
\quad \mathcal{R}\,e_1\,v_1 \to \mathcal{R}\,e_2\,v_2 \to \mathcal{R}\,(\texttt{Join}\,e_1\,e_2)\,(v_1, v_2).
\end{array}$$

A run can be either:

- the run of a `Ret` of a pure value v;
- the run of a `Call` of a command c that received an answer a from the environment, of type Effect.answer $E\,c$;
- the run of a `Let` of a computation e_1 and a function e_2 applied to the result of e_1, given by a run of e_1 and a run of $e_2\,v_1$;
- the run of a `Join` of two computations e_1 and e_2, given by a couple of runs for e_1 and e_2. Nothing is said about the interleaving of e_1 and e_2, we only observe the result of each computation.

Interestingly, a run can be viewed as the combination of two programs, a computation and its environment, with, by construction, calls and answers occurring at the same points and of compatible types. This idea will help us to define use cases, viewing the specification of a computation as a compatible program representing its environment.

7.5.3 Comparison

The semantic of the traces and the semantic of the runs are equivalent in the sense that we can define two bijections:

$$\begin{cases}
\texttt{of_run} : \forall\, E\,A\,(e : \mathcal{C}\,E\,A)\,(v : A)\,(r : \mathcal{R}\,e\,v),\\
\quad \{t : \mathcal{T}\,E \mid x \xrightarrow{t} v\}\\
\texttt{to_run} : \forall\, E\,A\,(e : \mathcal{C}\,E\,A)\,(t : \mathcal{T}\,E)\,(v : A),\\
\quad e \xrightarrow{t} v \to \mathcal{R}\,e\,v
\end{cases}$$

which convert a run to an accepted trace or an accepted trace to a run.

From a theoretical point of view, the run semantic is less satisfactory because a run is both a trace (which is a data type) and a proof of validity of a trace (which is a proposition, depending on both a trace and a computation). However, as we will see, the runs are often more practical to use.

7.6 Formal use cases

A USE CASE IS USUALLY PRESENTED though some textual or graphical presentations using a human language and metaphors. Formally, we define a use case as a set of list of interactions between the program and its environment, together with a set of program results for each possible list of interactions.

7.6.1 Definitions

We define a use case as a set of pairs of traces and results.

Definition 30 (Use cases)**.** *For an effect declaration E and an output type A, the type of the use cases $\mathcal{U}\,E\,A$ for the computations of type $\mathcal{C}\,E\,A$ is the type of the propositions over the traces of type $\mathcal{T}\,E$ and the values of type A:*

$$\mathcal{U}\,E\,A := \mathcal{T}\,E \times A \to \mathrm{Prop}$$

In informal definitions of use cases, the program result is often left unspecified. This is due to the fact that most programs verified with the use case technique live in $\mathcal{C}\,E\,\mathrm{unit}$, so the program result can only be the unique unit value. The specification of the program result is of particular importance when we want to compose the verification of sub-programs with a return type different from `unit`.

Definition 31 (Valid use cases)**.** *A computation e of type $\mathcal{C}\,E\,A$ verifies a use case u when:*

$$\forall\,(t : \mathcal{T}\,E)\,(v : A),\, u\,(t, v) \to e \xrightarrow{t} v$$

This means that the computation e must accept all the traces and output values described by the use case u. As an equivalent[11] way to define a use case, we can give a couple of a parametrized trace and a parametrized output value.

Definition 32 (Parametrized use cases)**.** *A parametrized use case for an effect declaration E and a return type A is an element of the type:*

$$\mathcal{U}\,E\,A := \{P : \mathrm{Type}\ \&\ ((P \to A) \times (P \to \mathcal{T}\,E))\}$$

A parametrized use case is formed of three elements:

- a type of parameters P;

[11] More generally, in Coq, a subset of a type A described by a parameter type P and a function $f : P \to A$ can be viewed as the predicate:

$$\begin{array}{rcccl} \varphi & : & A & \to & \mathrm{Prop} \\ & & a & \mapsto & \exists\,(p : P),\, f\,p \end{array}$$

Reciprocally, a subset given by a predicate $\varphi : A \to \mathrm{Prop}$ can be described by the parameter type:

$$P = \{a : A \mid \varphi\,a\}$$

and the canonical projection from P to A.

1. the program displays "What is your name?" on the terminal;
2. the program asks to read a new line;
3. the user answers by the string *name*;
4. the program displays a message containing the string *name*.

Figure 7.7: Informal use case of the computation `your_name`.

- a function from the parameters to the output values;
- a function from the parameters to the traces.

Definition 33 (Valid parametrized use cases)**.** *We say that a computation e verifies a parametrized use case $u = (P, v, t)$ if:*

$$\forall\, (p : P),\ e \xrightarrow{t\, p} v\, p$$

We introduce an preorder relation in order to compare two use cases.

Definition 34 (Use case generalization)**.** *A use case $u_1 = (P_1, v_1, t_1)$ generalizes a use case $u_2 = (P_2, v_2, t_2)$ if:*

$$\forall\, p_2 : P_2,\ \exists p_1 : P_1,\ \begin{cases} v_1\, p_1 = v_2\, p_2 \\ t_1\, p_1 =_{\mathsf{CO}} t_2\, p_2 \end{cases}$$

We formalize here the relation *generalize* of the UML use case diagrams. This relation is useful to specialize a use case on a more concrete and more understandable case. Due to some technicalities with the co-induction, we use the extensional and co-inductive equality $=_{\mathsf{CO}}$ instead of the standard Coq equality for the traces.

7.6.2 Illustration

We will use the notation:

$$\texttt{tlet!}\ t_1\ \texttt{in}\ t_2 \quad := \quad \texttt{TLet}\, t_1\, t_2$$

We present on the Figure 7.7 an informal use case of the computation `your_name` of the Figure 7.6 (page 123). We use the parametrized definition to formalize this use case. The computation `your_name` returns a value of type `unit`, so the specification of the output value is trivial. Using a type of parameters `string` for the variable *name*, we formalize the expected traces on the Figure 7.8. We state that the trace of interactions between the program and the environment must start by a call to the printing function with the argument "What is your name?" (line 2). The environment answers the unit value () to this printing call. Then, once

```
1 Definition your_name_uc (name : string) : 𝒯 E :=
2   tlet! (TCall (Print "What is your name?") ()) in
3   tlet! (TCall ReadLine (Some name)) in
4   TCall (Print ("Hello " + name)) ().
```

Figure 7.8: A use case for the computation `your_name`.

this call is completed, the program must ask to read a new line on the terminal (line 3). To this call the environment answers `Some` *name*, that is to say the string *name* with no errors. Then, the program must terminate by printing the message "Hello " + *name* (line 4).

To validate this use case, we must check that the `your_name` computation respects the scenario described by the use case, that is to say that for each parameter *name*, the trace:

$$\text{your\_name\_uc } name : \mathcal{T}\, E$$

is an acceptable trace for the computation `your_name`. The proof is very straightforward, using the definition of the accepted traces for a computation. In Coq, we apply the constructors representing the rules of the accepted traces, having just one choice at each step. This proof follows the same shape as the use case itself.

We will now see how to merge the definition and the verification of the use cases, by defining the use cases over the runs and by using the shape of the verified computation as a guide.

7.7 Use cases over the runs

We will define a use case as a set of runs instead of a set of traces. The main difference between a run and a trace is that the definition of a run includes the validity of its trace, and thus depends on a computation.

7.7.1 Definition

Definition 35 (Use cases over runs)**.** *We define type of the parametrized use cases over the runs of a computation e of type $\mathcal{C}\, E\, A$ by*[12]*:*

$$\mathcal{U}_R\, e := \{P : \text{Type} \;\&\; \{f : P \to A \;\&\; \forall p : P, \mathcal{R}\, e\, (f\, p)\}\}$$

A use case is composed of three elements:

- a type of parameters P;

[12] We define the use cases over the runs in the source file `src/UseCase.v` available on github.com/coq-io/io.

- a purely functional function f of type $P \to A$ expressing the results of the evaluation of e;
- a run of the computation e, parametrized by P, of result $f\,p$ for each parameter p.

The definition of $\mathcal{U}_R$ depends on a computation e. This is not totally satisfactory, because the expression of a specification should not depend on the verified program (only its verification should). However, in practice, we often verify a single program for a single specification, so this is less a problem to have a specification depending on a specific program. Moreover, we can convert a use case in $\mathcal{U}_R\,e$ to a use case over the traces of type $\mathcal{U}\,E$ by using the bijection `of_run` (page 125), which transforms runs into accepted traces.

The potential advantages of defining a use case over the runs instead of the traces are two folds:

- a use case in $\mathcal{U}_R\,e$ is by construction verified by the computation e, so we do not need a separated validity proof;
- we can define the runs of the use case following the shape of the computation. To this end, we exploit the proof mode of Coq. We use this mode like a debugger, to step through the computation e while defining the runs.

7.7.2 Symbolic debugger

In order to simplify the definition of proof terms (which are raw λ-terms), Coq provides a proof mode. In this mode, we use tactics to define a term piece by piece. The goals represent the types of the remaining holes to be filled. The hypothesis represent the variables of the current environment. Coincidentally, this mode also simplifies the definition of a use case as a parametrized run. The pieces of the definition are the constructors of the run type. For a computation e, the goals are of the form $\mathcal{R}\,e'\,v$ where e' is the current part of the computation which we consider.

For example, for the procedure `your_name` of the Figure 7.6, we start the definition of the use case by stating:

$$\texttt{Definition}\ \text{your\_name\_uc}\ (\textit{name} : \text{string}) : \mathcal{R}\,\text{your\_name}\,().$$

meaning that we will define a run over the computation `your_name`, which is parametrized by the string *name*. Coq answers us:

1 subgoal
name : string
_____(1/1)
$\mathcal{R}$ your_name ()

which means that we have to give the definition of a run over the computation `your_name` given a parameter *name*. Because this computation starts with

a `do!` instruction (on line 2), which is syntactic sugar for a `Let`, we know that the run must start by a `RLet` and be a sequence of two runs. We start with the constructor `RLet`, without specifying its arguments:

```
eapply RLet.
```

Coq answers:

```
2 subgoals
name : string
_____________________________________(1/2)
R (Call (Print "What is your name?")) ?70
_____________________________________(2/2)
R (
  let! name := Call ReadLine in
  match name with
  | Some name ⇒ Call (Print ("Hello " + name))
  | None ⇒ Ret ()
  end)
  ()
```

We now stepped into the `do!` instruction. We must fill two runs, one for the left-hand side of the `do!`, the other one for the right-hand side.

The first goal is the run of a call. We precisely wanted to enforce the printing of a welcome message at the beginning of our use case. We fill the first goal with:

```
apply (RCall (Print "What is your name?") ()).
```

because we want the computation to print this message, to which the environment answers the unit value (). By typing, Coq confirms that this is a valid run for the current computation. We could also let the string "What is your name?" implicit by using the unification mechanism of Coq to infer it automatically:

```
apply (RCall (Print _) ()).
```

To fill the second goal, we apply the same strategy by stepping into the `let!` using `eapply RLet`. Coq replies:

```
2 subgoals
name : string
_____________________________________(1/2)
R (Call ReadLine) ?99
_____________________________________(2/2)
R (
  match ?99 with
  | Some name ⇒ Call (Print ("Hello " + name))
  | None ⇒ Ret ()
  end)
  ()
```

```
1 Definition your_name_uc (name : string)
2   : R your_name ().
3   eapply RLet. apply (RCall (Print _) ()).
4   eapply RLet. apply (RCall ReadLine (Some name)).
5   apply (RCall (Print (_ + name)) ()).
6 Defined.
```

Figure 7.9: Use case for `your_name` as a parametrized run.

The special value ?99 is unknown at this point. We fill the first goal by stating that the computation must ask for a new line, to what the environment answers the *name* string:

```
apply (RCall ReadLine (Some name)).
```

The Coq system both validates this run for the current computation and infers that ?99 is the return value of the call, so `Some` *name*. The second goal becomes:

```
1 subgoal
name : string
_____________________________________(1/1)
R (
  match Some name with
  | Some name ⇒ Call (Print ("Hello " + name))
  | None ⇒ Ret ()
  end)
  ()
```

which simplifies to:

```
1 subgoal
name : string
_____________________________________(1/1)
R (Call (Print ("Hello " + name)) ()
```

We conclude by requiring the printing of the name:

```
apply (RCall (Print (_ + name)) ()).
```

To terminate the definition we enter:

```
Defined.
```

We resume the full definition of this use case over the runs on the Figure 7.9. This definition has the same shape as the use case defined over the traces on Figure 7.8. However, we can replay this use case step by step to understand how the computation `your_name` reacts to the environment's answers. Moreover, we

defined this use case with the help of the computation `your_name`, to guide our reasoning and infer the string "What is your name?". Thus, the proof mode of Coq played the role of a *symbolic debugger* to explore the computation step by step and helped to construct the use case.

By using the notations:

$$\begin{cases} \texttt{rlet!}\ r_1\ \texttt{in}\ r_2 & := & \texttt{RLet}\ r_1\, r_2 \\ \texttt{rdo!}\ r_1\ \texttt{in}\ r_2 & := & \texttt{RLet}\ (A := \text{unit})\, r_1\, r_2 \end{cases}$$

we can also directly write the use case without using the proof mode:

```
1  Definition your_name_uc (name : string)
2    : R your_name () :=
3    rdo! (RCall (Print _) ()) in
4    rlet! (RCall ReadLine (Some name)) in
5    RCall (Print (_ + name)) ().
```

This form can be simpler to read, however it is not possible to play it step by step.

7.8 Important patterns

We will present some important patterns we use to define use cases.

7.8.1 Composition

Because use cases are defined in Coq, we can compose small use cases to form larger specifications as we compose small programs to form larger ones. For example, the `your_name` procedure of the Figure 7.6 could be split into two computations:

```
1  Definition get_name : C E (option string) :=
2    do! Call (Print "What is your name?") in
3    Call ReadLine.
4
5  Definition your_name' : C E unit :=
6    let! name := get_name in
7    match name with
8    | None ⇒ Ret ()
9    | Some name ⇒ Call (Print ("Hello " + name))
10   end.
```

with the computation `get_name` returning either `Some` *name* in case of success or `None` in case of error. We define one use case for each computation:

```
1  Definition get_name_uc (name : string)
2    : R get_name (Some name).
3    eapply Let. apply (RCall (Print _) ()).
4    apply (RCall ReadLine (Some name)).
5  Defined.
6
7  Definition your_name_uc' (name : string)
8    : R your_name' ().
9    eapply Let. apply (get_name_uc name).
10   apply (RCall (Print (_ + name)) ()).
11 Defined.
```

The use case `get_name_uc` returns a run of type:

$$\mathcal{R}\,\mathrm{get\_name}\,(\mathtt{Some}\,name)$$

saying that it applies the computation `get_name` which gives a result `Some` *name*. In the definition of `your_name_uc'`, we call this use case with the parameter *name* on line 9 to follow the call of the computation `get_name` in the procedure `your_name'`. The specification of the return value is crucial to be able to compose these use cases.

In UML diagrams of use cases, specifications are composed using the *include* and *extend* relationships. A use case u_1 includes a use case u_2 if the definition of u_1 calls the definition of u_2. A use case u_1 extends a use case u_2 if, under certain conditions, the definition of u_2 calls the definition of u_1. The notions of *include* and *extend* can both be formalized in Coq using a call to another use case. We use a `if` over some conditions in the definition of a use case to express the condition of the *extend* relationship.

7.8.2 Recursion and higher-order

The use of recursive and higher-order procedures are two key elements of the functional programming. The definitions of the computations and the use cases can naturally use these two programming schemes.

As an example, we define the effectful `map_seq` function which sequentially applies a procedure to each element of a list (Figure 7.10). The `map_seq` function is polymorphic in the effect declaration E and the types A and B. If the list l is empty, we return the empty list (line 5). If the list is composed of a head x and of a tail xs, we apply f to the head element and then iterate over the tail to return a new list of type list B (line 9).

A generic use case for the procedure `map_seq` should depend on a use case for the parameter f, with a type of the form:

$$\mathcal{R}\,(f\,x)\,y$$

```
1  Fixpoint map_seq E A B
2    (f : A → C E B) (l : list A)
3    : C E (list B) :=
4    match l with
5    | [] ⇒ Ret []
6    | x :: xs ⇒
7      let! y := f x in
8      let! ys := map_seq f xs in
9      Ret (y :: ys)
10   end.
```

Figure 7.10: The recursive `map_seq` procedure.

for some relations between x, the argument of f, and its result y. We decide to relate the result and the argument of f by expressing both as functions of a parameter z:

$$\mathcal{R}\,(f\,(x\,z))\,(y\,z)$$

A use case for `map_seq` repeats the use case for the function f each time this function is applied to an element of the list. The full type for our generic use case for `map_seq` is:

$$\begin{array}{l}\text{map\_seq\_uc} : \forall\, E\,A\,B\,C\,(f : A \to \mathcal{C}\,E\,B)\\ \quad (l : \text{list}\,C)\,(x : C \to A)\,(y : C \to B)\\ \quad (u_f : \forall\, z,\, \mathcal{R}\,(f\,(x\,z))\,(y\,z)),\\ \quad \mathcal{R}\,(\text{map\_seq}\,f\,(\text{map}\,x\,l))\,(\text{map}\,y\,l)\end{array}$$

This use case is polymorphic in the effect declaration E and the types A, B and C. We use a list of parameters l to relate the list of arguments of `map_seq` ($\text{map}\,x\,l$, where `map` is the purely functional mapping of a function over a list) to the list of its results, $\text{map}\,y\,l$. Note that this list l is not the list l of the procedure `map_seq`. This use case is parametrized by a use case u_f over the function f.

We define the use case `map_seq_uc` by recursion, following the structure of `map_seq`. We reason by case on the list of parameters l:

```
destruct l as [|z zs].
```

If the list l is empty, the function `map_seq` returns the empty list with no effectful operations. Thus, we have a use case with no interactions:

```
apply RRet.
```

If the list l is composed of a head z and a tail zs, the function `map_seq` calls in sequence f on $x\,z$ and `map_seq` on $\text{map}\,x\,zs$. We apply in sequence the respective use cases:

```
eapply RLet. apply (u_f z).
eapply RLet. apply (map_seq_uc f zs x y u_f).
```

```
1  Fixpoint map_seq_uc E A B C (f : A → C E B)
2    (l : list C) (x : C → A) (y : C → B)
3    (u_f : ∀ z, R (f (x z)) (y z)) {struct l}
4    : R (map_seq f (map x l)) (map y l).
5    destruct l as [|z zs].
6    – apply RRet.
7    – eapply RLet. apply (u_f z).
8      eapply RLet. apply (map_seq_uc f zs x y u_f).
9      apply RRet.
10 Defined.
```

Figure 7.11: Use case for the procedure `map_sec`.

and conclude by the use case for the purely functional expression on line 8 of `map_seq`:

```
apply RRet.
```

We resume the full definition of `map_seq_uc` in the Figure 7.11.

7.8.3 Concurrency

We can transform the sequential procedure `map_seq` into a concurrent one by using the `Join` operator. Instead of applying the function f sequentially to each element of the list, we apply it concurrently on all the elements:

```
1  Fixpoint map_par E A B (f : A → C E B) (l : list A)
2    : C E (list B) :=
3    match l with
4    | [] ⇒ Ret []
5    | x :: xs ⇒
6      let! (y, ys) := Join (f x) (map_par f xs) in
7      Ret (y :: ys)
8    end.
```

On the line 6, we concurrently evaluate f on the head element x and on the tail elements xs of the list l.

We define a use case for the function `map_par` by adapting the previous use case of `map_seq`. Instead of playing sequentially a use case of f on each element

```
1  CoFixpoint say_hi : C E unit :=
2    do! Call (Print "Say hi:") in
3    let! s := Call ReadLine in
4    match s with
5    | None ⇒ Ret ()
6    | Some s ⇒
7      if s == "hi" then
8        Ret ()
9      else
10       say_hi
11   end.
```

Figure 7.12: Computation asking for the word *"hi"*.

of the list, we play a use case of f concurrently using the operator `RJoin`:

```
1  Fixpoint map_par_uc E A B C (f : A → C E B)
2    (l : list C) (x : C → A) (y : C → B)
3    (u_f : ∀ z, R (f (x z)) (y z)) {struct l}
4    : R (map_par f (map x l)) (map y l).
5    destruct l as [|z zs].
6    – apply RRet.
7    – eapply RLet. apply (
7        RJoin (u_f z) (map_par_uc f zs x y u_f)).
9      apply RRet.
10 Defined.
```

We notice that, in this use case, the procedure `map_par` associates the argument:

$$\mathrm{map}\, x\, l$$

to the result:

$$\mathrm{map}\, y\, l$$

as for the sequential version `map_seq`, even if the concurrent executions of the function f are now interleaved. This is because we specify the function f with the use case u_f, which depends on the parameter z but not the interleaving of f. This specification of `map_par` is relevant only if the scheduling of the applications of f does not matter.

7.8.4 Non-termination

We define a potentially non-terminating computation asking for the word *"hi"* until it gets it (Figure 7.12). This computation is defined co-inductively because it may not terminate (hence the `CoFixpoint` keyword on the first line).

Given a stream (an infinite list) of strings different from *"hi"*, we construct the non-terminating use case consisting of answering each element of

```
1 CoFixpoint say_hi_uc
2   (l : stream{s : string | s ≠ "hi"}) : R say_hi ().
3   destruct l as [[s H] l].
4   eapply RLet. apply (RCall (Print _) ()).
5   eapply RLet. apply (RCall ReadLine (Some s)).
6   rewrite (string_not_eq H).
7   apply (say_hi_uc l).
8 Defined.
```

Figure 7.13: Non-terminating use case for the computation `say_hi`.

this stream (Figure 7.13). The definition is slightly simplified. We define the function `say_hi_uc` by co-induction. We consume the stream parameter l on line 3 as a triple (s, H, l), with s the first string of the stream, H a proof that s is different from the message *"hi"*, and l the tail of the stream. On the lines 4 and 5, we run the print operation and answer the string s. We use the hypothesis H on line 6 to reduce to the `else` branch of the `if` of the computation. We conclude by recursing on the `say_hi_uc` use case.

7.9 Definitions of effects

We declare an effect by giving its signature, described by the type `Effect`. We will see different forms of definitions of effects, in order to modelize a use case environment or to implement a computation.

7.9.1 Effect refinement

The idea of the effect refinement is to program the computations and the use cases using some higher-level effect E_1, on top of a system where the calls are declared by some lower-level effect E_2.

Definition 36 (Effect refinement)**.** *An* effect refinement *between the effects E_1 and E_2 is a (purely functional) function φ of type:*

$$\varphi : \forall\, (c : \text{Effect.command}\, E_1),\ \mathcal{C}\, E_2\, (\text{Effect.answer}\, E_1\, c)$$

A refinement is an interpretation of the commands of E_1 using the commands of E_2. From a refinement, we derive by co-induction a compilation function[13] from the computations using E_1 to the computations using E_2 (Figure 7.14). This compilation replaces each command call by the application of the refinement φ (line 4).

[13] We define the compilation of a computation with an effect refinement in the source file `src/Evaluate.v` available on github.com/coq-io/evaluate.

```
1  CoFixpoint compile {A} (e : C E1 A) : C E2 A :=
2    match e with
3    | Ret v ⇒ Ret v
4    | Call c ⇒ φ c
5    | Let e1 e2 ⇒
6      let! v1 := compile e1 in
7      compile (e2 v1)
8    | Join e1 e2 ⇒ Join (compile e1) (compile e2)
9    end.
```

Figure 7.14: Compilation of a computation with an effect refinement.

Under some conditions, a use case over a computation of effect E_1 can be transported to a use case over computations of effect E_2.

Definition 37 (Run of refinement)**.** *A run of a refinement φ between E_1 and E_2 is a function r_φ of type:*

$$r_\varphi : \forall\, c\, (a : \text{Effect.answer}\, E_1\, c),\ \mathcal{R}\, (\varphi\, c)\, a$$

For any couple (c, a) of command and answer in E_1, this function explicits a run (in E_2) of the refinement of c yielding the answer a. Given a runnable refinement φ, we define by co-induction over the runs a function[14]:

$$\text{compile\_run} : \forall\, (e : \mathcal{C}\, E_1\, A)\, (v : A),\ \mathcal{R}\, e\, v \to \mathcal{R}\, (\text{compile}\, \varphi\, e)\, v$$

which transports any run r over a computation e with effect E_1 and result v to a run in E_2 of the compilation of e according to φ and with the same result v. Using this function `compile_run`, we can write use cases in E_1 and compile them to use cases in E_2.

In practice the effect refinement is useful to abstract a lower-level API, by hiding technicalities like the parsing, the pretty-printing of data or the raising of exceptions in the function φ. Instead of using the effect refinement, we could just replace in our computations each call of a command c by the expression $\varphi\, c$. However, by doing so, we would lose the documentation provided by the explicit declaration of the higher-level effect, we would not be able to write runs without defining the function r_φ, and we would miss the ability to provide alternative effect definitions. Indeed, as we will see, we can use alternative effect definitions to model and specify the program environment.

7.9.2 Modelization of the environment

In UML use case diagrams, the environment of a program is split into different *actors* playing different roles, in order to help the understanding. For example, we can partition the environment of the shell example of the Figure 7.1

[14] We define the compilation of a run with an effect refinement in the file `src/Evaluate.v` available on github.com/coq-io/evaluate.

into three actors: the user, the authentication process and the system running the commands. We formalize this partition by declaring the effect E of the shell environment as the effect composition (see definition 25) of the three effects:

$$E \;:=\; E_{\text{user}} \uplus E_{\text{auth}} \uplus E_{\text{commands}}$$

with one effect per actor.

We can model many actors as handlers with a state.

Definition 38 (State model)**.** *For two effects E_1 and E_2 and a state type S, we say that the handler φ and the state union $\cup$ form a* state model *of E_1 in E_2 if they are of types:*

$$\begin{cases} \varphi : \forall\, c,\;\; S \to \mathcal{C}\, E_2\, (\text{Effect.answer}\, E_1\, c \times S) \\ \cup : S \to S \to S \end{cases}$$

The handler φ interprets the commands of E_1 into commands of E_2 by using a state s of type S, and returns the answer of the command and the updated state. We model a `join` operation by duplicating the initial state and merging it at the end of the `join` using the state union operator $\cup$. For the actors which can be modeled in that way, this simplifies the reasoning about concurrent programs because there is no state sharing.

By co-induction over the computations, we deduce from a state model $(\varphi, \cup)$ a function `compile`[15] (Figure 7.15) in order to model a computation of type:

$$\mathcal{C}\, E_1\, A$$

by a computation with a state of type:

$$S \to \mathcal{C}\, E_2\, (A \times S)$$

Then, we can define the use cases for a computation e over its compilation `compile` e. As an illustration, for the shell, to model the commands actor E_{commands} with the two following shell commands:

- `AddUser` (*name* : string) (*description* : string) : unit to add a user in the users database,
- `GetUser` (*name* : string) : option string to get the description of a user (if it exists),

we define a state model from:

$$E_1 \;:=\; E_{\text{user}} \uplus E_{\text{auth}} \uplus E_{\text{commands}}$$

to:

$$E_2 \;:=\; E_{\text{user}} \uplus E_{\text{auth}}$$

[15] We define the compilation of a computation using a state model in the file `src/Evaluate.v` available on github.com/coq-io/evaluate.

```
1  CoFixpoint compile {A} (e : C E1 A) (s : S)
2    : C E2 (A × S) :=
3    match e with
4    | Ret v ⇒ Ret (v, s)
5    | Call c ⇒ φ c s
6    | Let e1 e2 ⇒
7      let! (v1, s) := compile e1 s in
8      compile (e2 v1) s
9    | Join e1 e2 ⇒
10     let! ((v1, s1), (v2, s2)) :=
11       Join (compile e1 s) (compile e2 s) in
12     Ret ((v1, v2), s1 ∪ s2)
13   end.
```

Figure 7.15: Compilation of a computation using a state model.

1. start with any users database;
2. add the user *name* with the description *description* and get an empty answer;
3. ask for the user *name* and get the description *description*;
4. continue the loop with another use case for commands.

Figure 7.16: Informal use case for the commands of the shell.

We use a state type $S := \text{list}\,(\text{string} \times \text{string})$, the association lists of names to descriptions, a state union:

$$l_1 \cup l_2 \;:=\; \text{assoc\_union}\, l_1\, l_2$$

the union of two association lists, and a handler φ defined by:

$$\begin{cases} \varphi\,(\texttt{AddUser}\,n\,d)\,s & := & \texttt{Ret}\,((), \text{assoc\_add}\,s\,n\,d) \\ \varphi\,(\texttt{GetUser}\,n)\,s & := & \texttt{Ret}\,(\text{assoc\_get}\,s\,n, s) \end{cases}$$

and for the commands of E_{user} or E_{auth}:

$$\varphi\,c\,s \;:=\; \texttt{let!}\; a := \texttt{Call}\,c\;\texttt{in}\;\texttt{Ret}\,(a, s)$$

The functions `assoc_add` and `assoc_get` respectively add a element into an association list and get the value of an element. Then we can formalize and verify use cases over the state model compilation of the commands loop of the shell. We give an example of informal use case we can formalize on the Figure 7.16.

7.9.3 Exceptions

Exceptions are a convenient programming concept to separate the treatment of exceptional cases from the normal control flow of a program. We will see how to implement exceptions using the computations.

Definition 39 (Error handler)**.** *For two effects E_1 and E_2, an error type $\mathcal{E}$ and an error union $\cup$, we say that φ is an* error handler *from E_1 to E_2 if the following typing judgments hold:*

$$\begin{cases} \varphi : \forall\, c,\ \mathcal{C}\, E_2\, (\text{Effect.answer}\, E_1\, c + \mathcal{E}) \\ \cup : \mathcal{E} \to \mathcal{E} \to \mathcal{E} \end{cases}$$

A command raising an exception is a command with an empty answer type (like the type `False`). As an illustration, for some error type $\mathcal{E}$, we declare the effect E_1 as the commands:

$$\texttt{Raise}\, (\epsilon : \mathcal{E})$$

with empty answers. An error handler in the terminal effect (see Figure 7.5, page 120) can be:

$$\begin{array}{l} \varphi\, (\texttt{Raise}\, \epsilon) := \\ \quad \texttt{do!}\ \texttt{Call}\, (\texttt{Print}\ \text{"An error occurred."})\ \texttt{in} \\ \quad \texttt{Ret}\, (\texttt{inr}\, \epsilon) \end{array}$$

where `inr` is the right injection:

$$\begin{cases} \texttt{inl} : \alpha \to \alpha + \beta \\ \texttt{inr} : \beta \to \alpha + \beta \end{cases}$$

For an error handler φ, we define a compilation function[16] over the computations by applying the error handler (Figure 7.17). We use the error union $\cup$ (on line 17) to collect all the possible errors raised by two concurrent threads launched by a `join` operator. Because we type the exceptions in the effects of the computations, we are sure not to forget to catch errors, what provide a safe framework to handle the exceptions. We remark that it is impossible to write a use case for a computation e raising an exception before to compile e, that is to say before defining how to handle the exception. Indeed, there are no traces or runs for the call of a command with an empty answer type.

7.9.4 Algebraic handlers

One of the idea of the algebraic effects is to define a generic shape of composable effect handlers. We will see how this kind of handlers can be used over the computations.

[16] We define the compilation of a computation with an error handler in the source file `src/Evaluate.v` available on github.com/coq-io/evaluate.

```
1  CoFixpoint compile {A} (e : C E1 A)
2    : C E2 (A + E) :=
3    match e with
4    | Ret v ⇒ Ret (inl v)
5    | Call c ⇒ φ c
6    | Let e1 e2 ⇒
7      let! v1 := compile e1 in
8      match v1 with
9      | inl v1 ⇒ compile (e2 v1)
10     | inr ε1 ⇒ Ret (inr ε1)
11     end
12   | Join e1 e2 ⇒
13     let! v := Join (compile e1) (compile e2) in
14     match v with
15     | (inl v1, inl v2) ⇒ Ret (inl (v1, v2))
16     | (inr ε, inl _) | (inl _ , inr ε) ⇒ Ret (inr ε)
17     | (inr ε1, inr ε2) ⇒ Ret (inr (ε1 ∪ ε2))
18     end
19   end.
```

Figure 7.17: Compilation of a computation using an error handler φ.

Definition 40 (Effect handler)**.** *For an effect E, a resource type S and an implementation M : Type → Type, we say that φ implements an* effect handler *if it has the type*[17]*:*

$$\varphi : \forall\, A\, (c : \text{Effect.command}\, E),\\ S \to (\text{Effect.answer}\, E\, c \to S \to M\, A) \to M\, A$$

We apply φ to a command, a state and a continuation k expecting the answer to the command and an updated state. Unfortunately, we cannot directly define a compilation function using an algebraic effect handler as before, for two reasons:

- the algebraic effect handlers evaluate the `Let` case by applying the current continuation, what would yield to an incorrect co-inductive definition. Indeed, co-inductive definitions must be productive;
- there are no direct equivalents of the `Join` operator in the (standard) theory of algebraic effects.

Using a restricted definition of the computations, defined by induction and without the `Join` operator, we define a compilation function[18] by applying an

[17] We define the type of an effect handler in the source file `src/Evaluate.v` available on github.com/coq-io/evaluate.

[18] We define the compilation of a computation with an algebraic effect handler in the file `src/Evaluate.v` available on github.com/coq-io/evaluate.

```
1  Fixpoint compile {A B} (e : C E A) (s : S)
2    (k : A → S → M B) : M B :=
3    match e with
4    | Ret v ⇒ k v s
5    | Call c ⇒ φ c s k
6    | Let e1 e2 ⇒
7      compile e1 s (fun v1 s ⇒
8      compile (e2 v1) s k)
9    end.
```

Figure 7.18: Compilation function for an algebraic handler.

algebraic handler φ (Figure 7.18). On line 5 we use the handler φ to evaluate the commands with the current continuation k. On lines 7 and 8, we sequentially compose the evaluation of e_1 and e_2 by representing the evaluation of e_2 as a continuation for e_1. The implementation type M can be any function over types, including a monad or a computation type $\mathcal{C}\,E'$ for some effect E'.

7.10 Related work

THE FORMALIZATION OF THE SPECIFICATIONS by use cases has already been done in other formalisms. The UML use case diagrams or the sequence diagrams, albeit informal, provide a first step toward the formalization of the use cases. Various works formalized these diagrams in formally defined languages, like in the language Z or the Petri nets [57]. However, as far as we know, we present the first formalization of the use cases in type theory.

Symbolic execution of programs [37] has been intensively studied, with the aim to extend the code coverage of program testing or with the aim to build symbolic debuggers [31]. Our method of definition of use cases (as runs using the proof mode of Coq) can be viewed as a form of interactive symbolic execution. We use the symbolic execution facilities of Coq to debug and verify the use cases.

The Ynot project studied the use of Coq both as a programming and specification language, by using a monad to represent impure operations. This monad allows to write and specify programs with pointers and mutable variables with a *Hoare*-logic and separation-logic [53] reasoning. The monad was extended to reason about sequential programs with inputs–outputs in order to certify a web application [42]. This work considers the formalization of trace invariants but not the formalization of use cases.

The algebraic effects and handlers [51] are a generic framework to write programs in a functional language, with explicit effects and in a compositional way. Various kinds of effects can be expressed with the algebraic effect handlers. The dependently-typed programming language Idris [7] implements these algebraic

effects [8]. In our approach, we explicitly separate the declaration of effects from their definitions, instead of parameterizing the computations with effect implementations. We believe that this offers more flexibility for the definition of effects: for example, we can use algebraic effect handlers but not only. To the best of our knowledge, specification by use cases over algebraic effects has not been studied yet.

7.11 Conclusion

We have presented the notion of *concurrent computations*, which are concurrent programs with inputs–outputs defined in the purely functional language Coq. We have defined a semantic for these computations, expressed using *traces* or using *runs*. The runs can be viewed as concurrent programs with inputs–outputs, with correct by construction interactions with a computation. By using the runs, we formally defined the use case specifications for the computations, expressing them as well-typed parametrized tests. Thanks to the proof mode of Coq and the formulation of the runs, the use cases can be written, debugged and proven correct in an interactive manner.

In the future, we would like to extend the class of programs handled by this framework. In particular, we would like to investigate the applicability of this method to programs defined as a set of concurrent actors exchanging messages. We would also like to investigate the question of the completeness of a specification by use cases. Given some constraints about the execution environment, how should we prove that a set of use cases actually covers all the possible execution scenarios?

Chapter 8

Blocking Computations

8.1 Abstract

Deadlocks are a common source of bugs in concurrent programs. They are hard to find and to reproduce because they can happen in a non-deterministic manner. Many methods have been proposed to automatically prevent deadlocks, like better programming practices or model checkers.

We present an expressive and generic framework in type theory to write and run concurrent and interactive higher-order programs with blocking operators. On top of this framework, we define a certified checker to generate a formal proof of deadlock-freedom for valid programs. The concurrent programs, the semantic and the checker are defined in the dependently typed proof and programming language Coq. This common platform is aimed to allow to combine the proofs generated by the automatic checker with proofs from other formal certification techniques.

8.2 Introduction

The design of correct concurrent applications with inputs–outputs is challenging. Indeed, due to the non-deterministic nature of the scheduling of events, concurrent programs are hard to test, verify or even reason about. However, concurrent programming is useful to write reactive and non-blocking programs, and to fully exploit the computing resources of modern multi-core architectures.

Many tools and techniques have been developed [43] to verify *abstract models* of concurrent programs. We can cite process algebras, session types, or model checking. Many of these tools are able to certify large programs of industrial use, but they rarely verify the actual implementation of the code. Instead, these tools usually verify a model of the program and the correctness of the modeling is left to the programmer [48].

Following the *software-proof co-design* approach, in which the code of a program and its correctness proofs are not disconnected but expressed in the same framework, we aim to design proof techniques applying directly to the implementation of concurrent programs. This requires a close connection between the programs, their specifications and their proofs. This connection is made possible in the proof and programming language Coq, a dependently typed, purely functional and higher-order language based on the CiC [18] (the Calculus of Inductive Constructions). Large software have been written and verified in the Coq system, like for example the language C's static analyzer Verasco [36]. However, in most cases, only the purely algorithmic and sequential parts are expressed and certified in Coq.

The concept of *monads* has been popularized by the Haskell community [49] as a convenient way to express concurrency and interactivity in a functional language. The design of a generic framework for concurrent and interactive monads is interesting because, in general, different monads cannot be easily combined. To verify programs written in this kind of frameworks, we believe this is also important to come up with a simple semantic and simple techniques, including decision procedures, taking advantage of the specificities of the Coq language.

In this chapter we will present[1]:

- the generic language of *blocking computations* to write higher-order concurrent and interactive programs in type theory. We will use an embedded domain-specific language in the dependently-typed programming language Coq (page 146);
- an operational semantic parametrized by the model of a list of potentially blocking commands (page 149), which we will use to express the property of deadlock-freedom (page 158);
- a simpler and semantically equivalent concurrent and interactive language, with the application to a proven-correct automatic checker validating the absence of deadlocks in a concurrent program (pages 159 and 152);
- some example of blocking concurrent primitives which can be represented by our framework (page 160).

8.3 Source language

We first introduce the notion of *blocking computations*, which are concurrent and interactive programs represented in type theory. These computations will be used to define the programs we want to run and verify.

[1] All the Coq formalizations of this chapter are available online on github.com/clarus/io-checker.

8.3.1 Commands

To interact with the system, an interactive program emits some *commands*. To these commands, the system may reply with an *answer* after an undefined amount of time. We introduce the type `Command.t` to represent these commands and the function `answer`:

$$\text{answer} : \text{Command.t} \to \text{Type}$$

to represent these commands and answers.

Definition 41 (Commands and answers)**.** *A* command *c is an element of the type `Command.t`. An* answer *a of c is an element of the type* answer c.

8.3.2 Computations

To represent interactive and concurrent programs in Coq, the *computations*, we introduce a set of unevaluated operators. The aim of a computation is to combine pure Coq code fragments and calls to the system into a more complex concurrent and interactive program.

Definition 42 (Blocking computation)**.** *The* blocking computations *returning a value of type A are described by the type $\mathcal{C}\,A$ defined by induction:*

$$\begin{array}{l}
\texttt{Inductive}\ \mathcal{C} : \text{Type} \to \text{Type} := \\
\mid \texttt{Return} : \forall\, A,\ \forall\, (x : A),\ \mathcal{C}\,A \\
\mid \texttt{Let} : \forall\, A\,B,\ \forall\, (x : \mathcal{C}\,A)\,(f : A \to \mathcal{C}\,B),\ \mathcal{C}\,B \\
\mid \texttt{Call} : \forall\, (c : \text{Command.t}),\ \mathcal{C}\,(\text{answer}\,c) \\
\mid \texttt{Join} : \forall\, A\,B,\ \forall\, (x : \mathcal{C}\,A)\,(y : \mathcal{C}\,B),\ \mathcal{C}\,(A \times B) \\
\mid \texttt{Choose} : \forall\, A,\ \forall\, (x_1 : \mathcal{C}\,A)\,(x_2 : \mathcal{C}\,A),\ \mathcal{C}\,A.
\end{array}$$

A blocking computation can be either:

- `Return` e, the pure expression e of type A. This expression can be any valid Coq expression, as long as it is of type A;
- `Let` $e_x\,e_f$, the application of the function e_f (returning a computation of type $\mathcal{C}\,B$) to the computation e_x returning a value of type A. This application is sequential, which means that the value v_x of the computation e_x has to be computed before the evaluation of $e_f\,v_x$;
- `Call` c, the call to the system with the command c. This call may be blocking, depending on how long the system is to answer. The answer of the system must be of type answer c;
- `Join` $e_x\,e_y$, the parallel evaluation of the computations e_x and e_y. This operator waits till the evaluation of both e_x and e_y is completed and returns the couple of results of e_x and e_y. The interleaving of the evaluation of e_x and e_y is unspecified;

Coq expression	Notation
`Return` e	$[e]$
`Let` $e_x\, e_f$	$e_x \gg\!= e_f$
`Let` $e_x\, (\_ \mapsto e_y)$	$e_x; e_y$
`Call` c	call c
`Join` $e_x\, e_y$	$e_x \parallel e_y$
`Choose` $e_1\, e_2$	$e_1 + e_2$

Table 8.1: Notations for the computations.

- `Choose` $e_1\, e_2$, the non-deterministic evaluation of either the computation e_1 or the computation e_2, returning the result of either e_1 or e_2. The choice of evaluation is done during the first call made by e_1 or e_2. If both calls are non-blocking, e_1 or e_2 can be evaluated. If one of the two expressions is blocking on the first call, the other is automatically executed. If both are blocking, `Choose` $e_1\, e_2$ is blocking too.

In the followings, we will use the notations given in the Table 8.1.

Examples The following computation concurrently runs two critical sections using a shared lock:

$$(\text{call lock}; s_1; \text{call unlock}) \parallel (\text{call lock}; s_2; \text{call unlock})$$

We manipulate to the shared lock through the two blocking commands `lock` and `unlock`. Assuming that the critical sections s_1 and s_2 do not access the lock, this computation is equivalent to the following:

$$(s_1; s_2) + (s_2; s_1)$$

and we execute either s_1 before s_2, or s_2 before s_1.

Despite the absence of recursion operators in the constructors of the computations, we can build recursive computations by using the `Fixpoint` operator of `Coq`. As an example we give the `prints` function:

```
Fixpoint prints (l : list string) : C unit :=
  match l with
  | . ⇒ [()]
  | s :: l ⇒ call (Print s); prints l
  end.
```

This function displays a list of strings using the call `Print` on each string of a list l. We use the operator ";" to sequence the calls to the printing operations.

$$
\begin{array}{rcl}
s & ::= & [\,] \\
 & | & s \gg\!= s \\
 & | & s \parallel s \\
 & | & s+ \\
 & | & +s
\end{array}
$$

Figure 8.1: Induction scheme for the value schedulings.

8.4 Semantic

We give an operational semantic to the language of the computations, and show how to interpret the calls to the commands. The rules to evaluate a computation either lead to a final value or lead to another computation by running a command. We guide the application of these rules using an explicit scheduling.

8.4.1 Values

We can reduce some computations to a value without executing any commands. For example, the computation:

$$12 \parallel (\texttt{false} + \texttt{true})$$

can be reduced non-deterministically to the values $(12, \texttt{false})$ or $(12, \texttt{true})$, by reducing $\texttt{false} + \texttt{true}$ either to `false` or to `true`. We will describe this choice by a *scheduling*.

Definition 43 (Value scheduling)**.** *We define the schedulings s for the values by the induction scheme on the Figure 8.1.*

A value scheduling can be $[\,]$ for a pure expression, $s_1 \gg\!= s_2$ for the binding of two computations, $s_1 \parallel s_2$ for a parallel evaluation, $s+$ for the evaluation on the left of a choice, or $+s$ for the evaluation on the right of a choice.

Definition 44 (Value reduction)**.** *We write $s \vdash e \Downarrow v$ to say that the computation e reduces to v using the value scheduling s. The reduction rules are the*

$$
\begin{array}{rcl}
S & ::= & \mathrm{call} \\
& | & S \gg\!= \\
& | & s \gg\!= S \\
& | & S \parallel \\
& | & s \parallel S \\
& | & \parallel S \\
& | & S \parallel s \\
& | & S+ \\
& | & +S
\end{array}
$$

Figure 8.2: Induction scheme for the command schedulings.

followings[2]*:*

$$\textsc{Ret}\ \frac{}{[\,] \vdash [e] \Downarrow e} \qquad \textsc{Let}\ \frac{s_x \vdash e_x \Downarrow v_x \qquad s_f : e_f\, v_x \Downarrow v_y}{s_x \gg\!= s_f \vdash e_x \gg\!= e_f \Downarrow v_y}$$

$$\textsc{Join}\ \frac{s_x \vdash e_x \Downarrow v_x \qquad s_y \vdash e_y \Downarrow v_y}{s_x \parallel s_y \vdash e_x \parallel e_y \Downarrow (v_x, v_y)} \qquad \textsc{ChooseLeft}\ \frac{s_1 \vdash e_1 \Downarrow v_1}{s_1+ \vdash e_1 + e_2 \Downarrow v_1}$$

$$\textsc{ChooseRight}\ \frac{s_2 \vdash e_2 \Downarrow v_2}{+s_2 \vdash e_1 + e_2 \Downarrow v_2}$$

8.4.2 Small-steps semantic for commands

Each step of the small-steps semantic corresponds to one execution of the call of a command.

Definition 45 (Command scheduling)**.** *The command schedulings S are inductively defined by the induction scheme on the Figure 8.2.*

A command scheduling can be `call` for the call of a command, $S \gg\!=$ for the evaluation of the left-hand side of a binding, $s \gg\!= S$ for the evaluation of the right-hand side of a binding once its left-hand side has been reduced to a value using a value scheduling, $S \parallel$ for the evaluation of a parallel composition starting by the left-hand side, $s \parallel S$ for the evaluation of a parallel composition once its left-hand side has been reduced to a value using a value scheduling (and similarly for the right-hand side of a parallel composition), $S+$ for the evaluation on the left of a choice and $+S$ for the evaluation on the right of a choice.

Definition 46 (Command reduction)**.** *We write $S \vdash e_1 \xrightarrow{c \to a} e_2$ to say that the computation e_1 reduces to the computation e_2 using the command scheduling S*

[2]We reason modulo the evaluation rules of `Coq`, assimilating each purely functional `Coq` expression to its value.

and executing the command c with an answer a. The reduction rules for the steps are the followings:

$$\textsc{Call}\ \frac{}{\text{call} \vdash \text{call}\ c \xrightarrow{c \to a} [a]} \qquad \textsc{Let}\ \frac{S_x \vdash e_x \xrightarrow{c \to a} e'_x}{S_x \gg\!= \vdash e_x \gg\!= e_f \xrightarrow{c \to a} e'_x \gg\!= e_f}$$

$$\textsc{LetDone}\ \frac{s_x \vdash e_x \Downarrow v_x \qquad S_f \vdash e_f\, v_x \xrightarrow{c \to a} e_y}{s_x \gg\!= S_f \vdash e_x \gg\!= e_f \xrightarrow{c \to a} e_y}$$

$$\textsc{JoinLeft}\ \frac{S_x \vdash e_x \xrightarrow{c \to a} e'_x}{S_x \parallel \vdash e_x \parallel e_y \xrightarrow{c \to a} e'_x \parallel e_y}$$

$$\textsc{JoinLeftDone}\ \frac{s_x \vdash e_x \Downarrow v_x \qquad S_y \vdash e_y \xrightarrow{c \to a} e'_y}{s_x \parallel S_y \vdash e_x \parallel e_y \xrightarrow{c \to a} e'_y \gg\!= (v_y \mapsto [(v_x, v_y)])}$$

$$\textsc{JoinRight}\ \frac{S_y \vdash e_y \xrightarrow{c \to a} e'_y}{\parallel S_y \vdash e_x \parallel e_y \xrightarrow{c \to a} e_x \parallel e'_y}$$

$$\textsc{JoinRightDone}\ \frac{S_x \vdash e_x \xrightarrow{c \to a} e'_x \qquad s_y \vdash e_y \Downarrow v_y}{S_x \parallel s_y \vdash e_x \parallel e_y \xrightarrow{c \to a} e'_x \gg\!= (v_x \mapsto [(v_x, v_y)])}$$

$$\textsc{ChooseLeft}\ \frac{S_1 \vdash e_1 \xrightarrow{c \to a} e'_1}{S_1 + \vdash e_1 + e_2 \xrightarrow{c \to a} e'_1} \qquad \textsc{ChooseRight}\ \frac{S_2 \vdash e_2 \xrightarrow{c \to a} e'_2}{+S_2 \vdash e_1 + e_2 \xrightarrow{c \to a} e'_2}$$

These reductions rules are pretty standard for a concurrent programming language, with maybe the exception of the rule for $e_x \parallel e_y$ when e_x reduces to a value v_x. In this case we state that $e_x \parallel e_y$ reduces directly to $e'_y \gg = (v_y \mapsto [(v_x, v_y)])$. We introduce this rule because this corresponds to what the implementation does and because it will simplifies the verification of our automatic checker.

8.4.3 Interpretation of commands

For now the commands are purely abstract, because we just assumed couples of well typed commands and answers to define the reduction rules. We will relate an answer a to its command c and explicit the blocking calls.

Definition 47 (Model)**.** *A* model *is a couple $\mathcal{M} = (\mathcal{S}, \varphi)$ with $\mathcal{S}$ a type and φ a function:*

$$\varphi : \forall\, (c : \text{Command.t})\, (s : \mathcal{S}),\ (\text{answer}\, c \times \mathcal{S}) \uplus \{\bot\}$$

An element s of type $\mathcal{S}$ represents a state of the environment of a computation. The function φ represents an evaluation function which, for a command c

c	s	$\varphi\ c\ s$
lock	false	(OK, true)
lock	true	$\bot$
unlock	false	$\bot$
unlock	true	(OK, false)

Table 8.2: Definition of the function φ.

and an environment s, returns an answer a and a new environment s', or $\bot$ if the call is blocking. We define the reduction of a computation in a model $\mathcal{M}$.

Definition 48 (Model reduction)**.** *Given a model $\mathcal{M}$, the reduction of a computation e to a computation e', using a command c in the state s and following the scheduling S, is defined by the single inference rule:*

$$\textsc{Model}\ \frac{\varphi\, c\, s = (a, s') \qquad S \vdash e \xrightarrow{c \to a} e'}{S \vdash_{\mathcal{M}} (e, s) \xrightarrow{c} (e', s')}$$

Example To represent a single global lock we define two commands:

$$\begin{array}{rcl} \text{Command.t} & ::= & \text{lock} \\ & | & \text{unlock} \end{array}$$

and a state $\mathcal{S} = \text{bool}$ representing the state of the lock (acquired or not). We define the function φ on the Table 8.2. The commands block trying to acquire an acquired lock or trying to release a released lock.

8.5 A simpler and equivalent language

We will define a concurrent and interactive language *choose*, simpler than the language of the *computations*. We will prove these two languages to be equally expressive. This more primitive language will provide us the basis to build and verify an automatic checker to verify deadlock-freedom.

8.5.1 Definition

The *choose* syntax is specified by the parametrized type $\mathcal{H}$.

Definition 49 (Choose language)**.** *The set of choose expressions returning a value of type A is represented by the type $\mathcal{H}\,A$, inductively defined by:*

$$\begin{array}{l} \texttt{Inductive}\ \mathcal{H}\,A : \text{Type} := \\ |\ \texttt{Return} : A \to \mathcal{H}\,A \\ |\ \texttt{Call} : \forall\,(c : \text{Command.t}),\ (\text{answer}\,c \to \mathcal{H}\,A) \to \mathcal{H}\,A \\ |\ \texttt{Choose} : \mathcal{H}\,A \to \mathcal{H}\,A \to \mathcal{H}\,A. \end{array}$$

A *choose* expression can be either:

- `Return` e, the pure `Coq` expression e of type A;
- `Call` $c\,h$, the call of the command c with the handler h. The handler will be called with the result of the command once it returns;
- `Choose` $e_1\,e_2$, the non-deterministic evaluation of either e_1 or e_2.

This language shares similarities with the language of the computations, but lacks the `Let` and the `Join` operators. Since we are in continuation-passing style, the `Call` operator calls a handler instead of returning an answer to the evaluation context. We will use the same notations $[e]$, call $c\ h$ and $e_1 + e_2$ to represent the choose expressions.

8.5.2 Semantic

Like for the language of computations, we will define an operational semantic with two kinds of evaluations rules, for the terms returning a value and for the terms calling a command.

Definition 50 (Choose scheduling)**.** *The choose-scheduling to compute a value are inductively defined by:*

$$\begin{array}{rcl} s & ::= & \cdot \\ & | & s+ \\ & | & +s \end{array}$$

There is just one kind of schedulings for the choose expressions. We will define the rules for the reductions to a value and for the reductions to another choose expression.

Definition 51 (Value reduction)**.** *We write $s \vdash e \Downarrow v$ to say that an expression e reduces to the value v using the scheduling s. The reduction to a value is defined by the following inference statements:*

$$\textsc{Ret}\ \frac{}{\cdot \vdash [e] \Downarrow e} \qquad \textsc{ChooseLeft}\ \frac{s_1 \vdash e_1 \Downarrow v_1}{s_1+ \vdash e_1 + e_2 \Downarrow v_1}$$

$$\textsc{ChooseRight}\ \frac{s_2 \vdash e_2 \Downarrow v_2}{+s_2 \vdash e_1 + e_2 \Downarrow v_2}$$

Definition 52 (Command reduction)**.** *We write $s \vdash e_1 \xrightarrow{c \to a} e_2$ to say that the expression e_1 reduces to the expression e_2 using the scheduling s and executing the command c with an answer a. We inductively define this reduction relation*

$$
\begin{array}{rcl}
\text{bind}\ [v_x]\, e_f & = & e_f\ v_x \\
\text{bind}\ (\text{call}\ c\, h)\, e_f & = & \text{call}\ c\, (a \mapsto \text{bind}\ (h\, a)\, e_f) \\
\text{bind}\ (e_1 + e_2)\, e_f & = & (\text{bind}\ e_1\, e_f) + (\text{bind}\ e_2\, e_f) \\
 & & \\
[v_x]\ {}^{\curvearrowleft}\!\!\parallel\ e_y & = & \text{bind}\ e_y\, (v_y \mapsto [(v_x, v_y)]) \\
\text{call}\ c\, h\ {}^{\curvearrowleft}\!\!\parallel\ e_y & = & \text{call}\ c\, (a \mapsto h\, a \parallel e_y) \\
(e_1 + e_2)\ {}^{\curvearrowleft}\!\!\parallel\ e_y & = & (e_1\ {}^{\curvearrowleft}\!\!\parallel\ e_y) + (e_2\ {}^{\curvearrowleft}\!\!\parallel\ e_y) \\
 & & \\
e_x\ \parallel^{\curvearrowright}\ [v_y] & = & \text{bind}\ e_x\, (v_x \mapsto [(v_x, v_y)]) \\
e_x\ \parallel^{\curvearrowright}\ \text{call}\ c\, h & = & \text{call}\ c\, (a \mapsto e_x \parallel h\, a) \\
e_x\ \parallel^{\curvearrowright}\ (e_{y_1} + e_{y_2}) & = & (e_x\ \parallel^{\curvearrowright}\ e_{y_1}) + (e_x\ \parallel^{\curvearrowright}\ e_{y_2}) \\
 & & \\
e_x \parallel e_y & = & (e_x\ {}^{\curvearrowleft}\!\!\parallel\ e_y) + (e_x\ \parallel^{\curvearrowright}\ e_y)
\end{array}
$$

Figure 8.3: Combinators over the choose expressions.

by the following rules:

$$\textsc{Call}\ \frac{}{\cdot \vdash \text{call}\ c\, h \xrightarrow{c \to a} h\, a} \qquad \textsc{ChooseLeft}\ \frac{s_1 \vdash e_1 \xrightarrow{c \to a} e_1'}{s_1 + \vdash e_1 + e_2 \xrightarrow{c \to a} e_1'}$$

$$\textsc{ChooseRight}\ \frac{s_2 \vdash e_2 \xrightarrow{c \to a} e_2'}{+ s_2 \vdash e_1 + e_2 \xrightarrow{c \to a} e_2'}$$

8.5.3 Equivalence

We will prove that the choose language, albeit simpler than the computations language, is semantically equivalent. This equivalence will be proven by first defining a compilation of both the computation expressions and schedulings to the choose expressions and schedulings. We also compile the schedulings to obtain a more informative proof.

Compilation of computations We compile the computations to the choose expressions. We do a transformation by continuation to eliminate the `Let` operator and we expand the parallel composition `Join` to a complete choice between the different possible interleavings of the threads using the `Choose` operator. Due to this interleaving, the resulting choose expression may be exponentially large compared to the original computation.

We define by induction over the choose expressions the functions on the Figure 8.3. The `bind` operator sequentially composes two choose expressions. The expression $e_x\ {}^{\curvearrowleft}\!\!\parallel\ e_y$ concurrently composes the choose expressions e_x and e_y by starting to evaluate a command in e_x if possible (for example, if e_x is not already some $[v_x]$). Similarly we define the expression $e_x\ \parallel^{\curvearrowright}\ e_y$ so that the parallel composition $\parallel$ can be expressed as a choice between the parallel composition

to the left and the parallel composition to the right. This definition has the advantage of being symmetric[3]. The Coq interpreter accepts these definitions, ensuring that these (recursive) functions are terminating.

We conclude by the definition of the compilation function ϕ:

$$\begin{array}{rcccl} \phi & : & \mathcal{C}\,A & \to & \mathcal{H}\,A \\ & & [v_x] & \mapsto & [v_x] \\ & & e_x \gg\!= e_f & \mapsto & \text{bind}\ (\phi\, e_x)\,(v_x \mapsto \phi\,(e_f\, v_x)) \\ & & \text{call}\ c & \mapsto & \text{call}\ c\,(a \mapsto [a]) \\ & & e_x \parallel e_y & \mapsto & \phi\, e_x \parallel \phi\, e_y \\ & & e_1 + e_2 & \mapsto & \phi\, e_1 + \phi\, e_2 \end{array}$$

Compilation of schedulings We define a compilation of the schedulings which transports a valid scheduling of an expression e to a valid scheduling of the compilation $\phi\, e$ (Figure 8.4). We first define a $s_1; s_2$ operator which basically sequences two choose schedulings. The function ϕ_s compiles a computation scheduling to values. The compilation of $s_x \gg\!= s_f$ is the sequencing of the compilations of s_x and s_f. The compilation of $s_x \parallel s_y$ makes an arbitrary choice[4] to first run s_x. This choice is not important because the computations reducing to values do not make observable calls. The compilation of the call schedulings ϕ_S is similar and does not make arbitrary choices. We compile a scheduling of a parallel composition to the scheduling of a choose operator, with a choice to the left or a choice to the right depending on the kind of parallel scheduling.

Similarly, we define a reverse compilation (Figure 8.5) which, given a choose scheduling and a computation e, returns a computation scheduling following the structure of e. The reverse compilation returns either the scheduling to a command, or a triple containing the scheduling to a value, the value itself and a remaining choose scheduling. This more generic form is required to have a recursive definition handling the $e_x \gg\!= e_f$ case. Some cases are not supposed to happen and thus will have no roles in the proof, like for example:

$$\begin{cases} \phi^{-1}\ (e_x \parallel e_y)\ \cdot & = & \text{call} \\ \phi^{-1}\ (e_1 + e_2)\ \cdot & = & \text{call} \end{cases}$$

The definition of the reverse compilation is more complex because the choose schedulings are less informative than the computation schedulings. Thus, we must use the structure of the original computation in order to recover the original scheduling.

[3] We could also define the parallel composition with:

$$\begin{array}{l} e_1 \| e_2 = e_1 \ \text{↰}\| \ e_2 + \\ \quad \text{bind}\,(e_2 \ \text{↰}\| \ e_1)((v_1, v_2)) \mapsto [(v_2, v_1)]) \end{array}$$

However, this definition is not symmetric and would make the proof of equivalence between the computations and the choose expressions more complex.

[4] The other choice being:

$$+(\phi_s\, s_y\, ;\, \phi_s\, s_x)$$

$$
\begin{array}{lcl}
; & : & s_{\mathcal{H}} \to s_{\mathcal{H}} \to s_{\mathcal{H}} \\
\cdot\,;\, s_f & = & s_f \\
s_1{+}\,;\, s_f & = & (s_1\,;\, s_f){+} \\
{+}s_2\,;\, s_f & = & {+}(s_2\,;\, s_f)
\end{array}
$$

$$
\begin{array}{lcl}
\phi_s \;:\; s_{\mathcal{C}} & \to & s_{\mathcal{H}} \\
[\,] & \mapsto & \cdot \\
s_x \gg\!= s_f & \mapsto & \phi_s\, s_x\,;\, \phi_s\, s_f \\
s_x \parallel s_y & \mapsto & (\phi_s\, s_x\,;\, \phi_s\, s_y){+} \\
s_1{+} & \mapsto & (\phi_s\, s_1){+} \\
{+}s_2 & \mapsto & {+}(\phi_s\, s_2)
\end{array}
$$

$$
\begin{array}{lcl}
\phi_S \;:\; S_{\mathcal{C}} & \to & s_{\mathcal{H}} \\
\mathrm{call} & \mapsto & \cdot \\
S_x \gg\!= & \mapsto & \phi_S\, S_x \\
s_x \gg\!= S_f & \mapsto & \phi_s\, s_x\,;\, \phi_S\, S_f \\
S_x \parallel & \mapsto & (\phi_S\, S_x){+} \\
s_x \parallel S_y & \mapsto & (\phi_s\, s_x\,;\, \phi_S\, S_y){+} \\
\parallel S_y & \mapsto & {+}(\phi_S\, S_y) \\
S_x \parallel s_y & \mapsto & {+}(\phi_s\, s_y\,;\, \phi_S\, S_x) \\
S_1{+} & \mapsto & (\phi_S\, S_1){+} \\
{+}S_2 & \mapsto & {+}(\phi_S\, S_2)
\end{array}
$$

Figure 8.4: Compilation from computation schedulings to choose schedulings.

$$
\begin{array}{lrcll}
\phi^{-1} : & \mathcal{C}\,A & \to\ s_{\mathcal{H}} & \to & (s_{\mathcal{C}} \times A \times s_{\mathcal{H}}) \uplus S_{\mathcal{C}} \\
& [v] & \mapsto\ s & \mapsto & (\cdot, v, s) \\
& \text{call } c & \mapsto\ s & \mapsto & \text{call} \\
& e_x \gg\!= e_f & \mapsto\ s & \mapsto & \texttt{match } \phi^{-1}\, e_x\ s \texttt{ with} \\
& & & & \quad \mid (s_x, v_x, s) \mapsto \\
& & & & \quad\ \texttt{match } \phi^{-1}\,(e_f\, v_x)\ s \texttt{ with} \\
& & & & \quad\ \mid (s_f, v_y, s) \mapsto (s_x \gg\!= s_f, v_y, s) \\
& & & & \quad\ \mid S_y \mapsto s_x \gg\!= S_y \\
& & & & \quad \mid S_x \mapsto S_x \gg\!= \\
& e_x \parallel e_y & \mapsto\ s & \mapsto & \texttt{match } s \texttt{ with} \\
& & & & \quad \mid \cdot \mapsto \text{call} \\
& & & & \quad \mid s + \mapsto \\
& & & & \quad\ \texttt{match } \phi^{-1}\, e_x\, s \texttt{ with} \\
& & & & \quad\ \mid (s_x, v_x, s) \mapsto \\
& & & & \quad\ \quad \texttt{match } \phi^{-1}\, e_y\, s \texttt{ with} \\
& & & & \quad\ \quad \mid (s_y, v_y, s) \mapsto (s_x \parallel s_y, (v_x, v_y), s) \\
& & & & \quad\ \quad \mid S_y \mapsto s_x \parallel S_y \\
& & & & \quad\ \mid S_x \mapsto S_x \parallel \\
& & & & \quad \mid\ + s \mapsto \\
& & & & \quad\ \texttt{match } \phi^{-1}\, e_y\, s \texttt{ with} \\
& & & & \quad\ \mid (s_y, v_y, s) \mapsto \\
& & & & \quad\ \quad \texttt{match } \phi^{-1}\, e_x\, s \texttt{ with} \\
& & & & \quad\ \quad \mid (s_x, v_x, s) \mapsto (s_x \parallel s_y, (v_x, v_y), s) \\
& & & & \quad\ \quad \mid S_x \mapsto S_x \parallel s_y \\
& & & & \quad\ \mid S_y \mapsto \parallel S_y \\
& e_1 + e_2 & \mapsto\ s & \mapsto & \texttt{match } s \texttt{ with} \\
& & & & \quad \mid \cdot \mapsto \text{call} \\
& & & & \quad \mid s + \mapsto \\
& & & & \quad\ \texttt{match } \phi^{-1}\, e_1\, s \texttt{ with} \\
& & & & \quad\ \mid (s_1, v_1, s) \mapsto (s_1+, v_1, s) \\
& & & & \quad\ \mid S_1 \mapsto S_1+ \\
& & & & \quad \mid\ + s \mapsto \\
& & & & \quad\ \texttt{match } \phi^{-1}\, e_2\, s \texttt{ with} \\
& & & & \quad\ \mid (s_2, v_2, s) \mapsto (+\, s_2, v_2, s) \\
& & & & \quad\ \mid S_2 \mapsto +\, S_2
\end{array}
$$

Figure 8.5: Compilation from choose schedulings to computation schedulings, following the structure of a computation.

Equivalence We introduce a labeled transitions system over the union of the computations and the choose expressions.

Definition 53 (Label)**.** *A* label l *is either* $\cdot$ *or* $c{\rightarrow}a$ *for a command* c *and an answer* a *of the corresponding type.*

Definition 54 (Labeled transition system)**.** *For a type* A*, we define the labeled transition system* $e \xrightarrow{l} e'$ *over* $\mathcal{C}\,A \uplus \mathcal{H}\,A$ *by the following rules:*

$$\text{C-Value}\ \frac{\exists\,(s : s_{\mathcal{C}}),\ s \vdash e \Downarrow v}{e \xrightarrow{\cdot} [v]} \qquad \text{C-Command}\ \frac{\exists\,(S : S_{\mathcal{C}}),\ S \vdash e \xrightarrow{c\rightarrow a} e'}{e \xrightarrow{c\rightarrow a} e'}$$

$$\text{H-Value}\ \frac{\exists\,(s : s_{\mathcal{H}}),\ s \vdash e \Downarrow v}{e \xrightarrow{\cdot} [v]} \qquad \text{H-Command}\ \frac{\exists\,(s : s_{\mathcal{H}}),\ s \vdash e \xrightarrow{c\rightarrow a} e'}{e \xrightarrow{c\rightarrow a} e'}$$

We are now ready to assert that the compilation function ϕ from the computations to the choose expressions is correct, in the sense that the compiled expressions are bisimilar to the original ones.

Theorem 9. *For a type* A*, the relation* R *over* $\mathcal{C}\,A \uplus \mathcal{H}\,A$ *defined by:*

$$R\ e_1\ e_2\ :=\ (e_2 = \phi\, e_1)$$

is a bisimulation over the labeled transition system $e \xrightarrow{l} e'$*.*

We prove this theorem using the explicit compilations of schedulings, by inductions following the inductive definitions of the compilations of schedulings. We verify this proof in the Coq proof system[5]. To avoid some technical difficulties with dependent types, we only verify the case in which the type of the answers is not dependent on the value of the commands. Thus the proof is not complete yet, but we think that adding dependent answers does not change the validity the bisimulation.

8.6 Automatic checker for deadlock-freedom

We define and prove correct a simple automatic checker for deadlock-freedom in the *computations*, by using the equivalent choose language.

8.6.1 Deadlock-freedom

We first define the notion of deadlocks in our settings. The following definition applies equally to the choose expressions.

[5] The Coq formalizations of this chapter are available online on github.com/clarus/io-checker.

```
(* Check that we can execute at least one step. *)
not_stuck s e :=
  match e with
  | [v] ↦ true
  | call c h ↦ (φ c s ≠ ⊥)
  | e1 + e2 ↦ not_stuck s e1 ∨ not_stuck s e2

(* Check that all non-blocked execution paths lead to a
   deadlock free expression. *)
explore s e :=
  match e with
  | [v] ↦ true
  | call c h ↦
    match φ c s with
    | (a, s') ↦ deadlock_free s' (h a)
    (* Since this path is blocked we have nothing to check. *)
    | ⊥ ↦ true
  | e1 + e2 ↦ explore s e1 ∧ explore s e2

deadlock_free s e :=
  not_stuck s e ∧ explore s e
```

Figure 8.6: Algorithm to check the deadlock-freedom.

Definition 55 (Deadlock-freedom)**.** *Given a model of commands $\mathcal{M} = (\mathcal{S}, \varphi)$, a computation x is said to be* deadlock-free *in the state s if the two following points are true:*

- *x can reduce to a value or another computation by executing a non-blocking command, and*
- *for each (non-blocking) reduction of the form:*

$$S \vdash_{\mathcal{M}} (x, s) \xrightarrow{c} (x', s')$$

x' is deadlock-free in the state s'.

8.6.2 Algorithm

We write a decision procedure to check if a choose expression is deadlock-free (Figure 8.6). We proceed by exploring the reachable states of the expression tree using the function φ of the model in order to evaluate each command. Along this exploration, we verify that each sub-expression is not stuck. An expression is not stuck if there exists a scheduling to a final value or to a non-blocking call. To check the deadlock-freedom of a computation x, we compile x to a choose expression and then run the decision procedure.

The decision procedure is terminating because its recursive definition is validated by the Coq proof system, which only accepts terminating functions. The termination is not completely obvious because we recurse on the handler $h\ a$ in the case "call $c\ h$" of the function `explore`. Indeed, this handler may return a choose expression syntactically larger than the initial expression.

We can remark that, since we explore all the possible execution paths, a deadlock-free choose expression always terminates to a value in any scheduling. By bisimulation, this result holds for the computations too. In particular, the non-termination of an infinite "while true" loop cannot be implicitly expressed as a computation, and one should proceed with the set of the finite loop unrollings.

8.6.3 Correctness

The `deadlock_free` procedure is correct.

Theorem 10. *Given a model of commands $\mathcal{M} = (\mathcal{S}, \varphi)$, a state s and a computation e, if:*

$$\text{deadlock\_free}\ s\ e\ =\ \texttt{true}$$

then e is deadlock-free in the state s.

We show the correctness of our decision procedure on the choose expressions. The correctness for the computations is then deduced by bisimulation. We separately prove the correctness of the `not_stuck` procedure by induction on x, showing that it decides if an expression can be reduced to a value or another expression. We prove the correctness of the `deadlock_free` procedure by induction on x, and then by a second nested induction on x for each possible reduction of the form:

$$S \vdash_{\mathcal{M}} (x, s) \xrightarrow{c} (x', s')$$

The proof is validated by the Coq proof checker, and can be found at the end of the file `src/Decide.v` in the Coq development[6].

8.7 Examples

We have presented the definition of locks (page 151). We will present the formalization of more synchronization primitives and the verification of some code examples. More details about the examples can be found in the Coq development.

[6] All the Coq formalizations of this chapter are available online on github.com/clarus/io-checker.

```
Definition shell (x : C A) : C A :=
  incr;
  x ≫= (y ↦
  decr;
  [y]).
```

Figure 8.7: Encapsulate a computation in a semaphore.

```
Fixpoint map_sem (f : A → C B)
  (l : list A) : C (list B) :=
  match l with
  | · ⇒ [·]
  | x :: l ⇒
    (shell (f x) ∥ map_sem f l) ≫=
    ((y, l) ↦ [y :: l])
  end.
```

Figure 8.8: Iterate over a list with at most n concurrent operations.

8.7.1 Semaphores

We model a semaphore of size n with the state $S_n := 0, 1, \ldots, n$. We use two operations `incr` and `decr` to atomically increment or decrement the state. We define the transition function φ as:

$$\begin{array}{rcccccl} \varphi & : & \text{Command.t} & \to & S_n & \to & (\cdot \times S_n) \cup \bot \\ & & \text{incr} & \mapsto & k & \mapsto & (\cdot, k+1) \text{ if } k \neq n, \text{else } \bot \\ & & \text{decr} & \mapsto & k & \mapsto & (\cdot, k-1) \text{ if } k \neq 0, \text{else } \bot \end{array}$$

The operations block if no more resources are available or if all resources were already released. As an example, we implement a generic `map_sem` function (Figure 8.8) which concurrently applies a function f on the elements of a list l with at most n parallel executions of the function f. We use the function `shell` (Figure 8.7) which encapsulates a computation in between an `incr` and a `decr` operations, so that the computation can be executed at most n times simultaneously.

8.7.2 Transactional memory

We implement a simple form of transactional memory to encode the problem of the n-philosophers. The decision procedure was able to verify the deadlock-freedom for up to 7 philosophers.

8.7.3 Message passing

We represent a message box of type A with a state $S_A := A \uplus \bot$, a blocking `send` when the box is full and a blocking `receive` when the box is empty:

$$\begin{cases} \text{send}(x) & : & s & \mapsto & (\cdot, x) & \text{if } s = \bot, \text{else } \bot \\ \text{receive} & : & s & \mapsto & (x, \bot) & \text{if } s \neq \bot, \text{else } \bot \end{cases}$$

8.8 Related work

Safe concurrent programming have been studied a lot for functional but non-dependently typed languages such as Haskell [50]. We try to build on this experience exploiting the extended type system of Coq.

The algebraic effects system [51] is a generic framework to represent effects as collections of handlers. The application of the algebraic effects to concurrency has been considered, for example to represent the operators of the CSP calculus [61]. However, as far as our knowledge goes, the algebraic effects do not permit a clean encoding of the concurrency operators yet.

The Idris language [7] is a dependently typed programming language implementing the algebraic effects system. This language also provides concurrency primitives, with the ability to compile down to the Erlang language for example [22], but as far as we know no formal semantics is given to these primitives. Edwin Brady and Kevin Hammond show how to give the definition of an embedded language in Idris were programs are restricted so that they are deadlock-free by construction [10], but this language is not as expressive as the language of the computations.

Various forms of the *Separation Logic* have been encoded in Coq as a state monad extended with pre/post-conditions and invariants. The *Hoare Type Theory* has been applied to the concurrent setting [46]. The FCSL framework [56] extends these techniques by providing powerful reasoning rules to verify fine-grained concurrent programs. This work focuses on the treatment of low-level (but higher-order) imperative algorithms acting on a shared heap; on the contrary, we avoid the programs manipulating memory pointers and try to work on higher-level primitives and runnable examples. However, the two concurrent languages seem to share many similarities, so building a formal relation between the two could be very interesting.

Finally, many static analysis tools for concurrent programming languages have been developed. We can cite for example the JavaPathFinder[7] tool which is a model checker for concurrent Java programs. In contrast with this model checker, we aim to provide a certified tool integrated into the general purposes proving platform Coq.

[7] The JavaPathFinder tool is available on babelfish.arc.nasa.gov/trac/jpf under Apache license.

8.9 Conclusion

We have presented the idea of *blocking computations*, which are concurrent and interactive programs written in the dependently typed language Coq. We give an operational semantic to the language of the computations to model various kinds of synchronization primitives. We have shown how to compile computations to a simpler but semantically equivalent language, and how to derive a verified decision procedure for a deadlock-freedom analysis.

On the future, we would like to investigate more the interactions and the composability of various proofs techniques usable for concurrent programs, in the settings of the language of computations. We would also like to study the ideas of the compilation to the choose language as a way to provide a certified runtime for the language of the computations[8].

[8] As suggested by one the reviewers of this chapter.

Chapter 9

Conclusion

9.1 What we have done

In this thesis we have studied various approaches aiming at using Coq as a programming language, guiding our developments by examples.

9.1.1 Concurrent computations

Our main development is the language of *concurrent computations* (chapter 7) to write programs with asynchronous inputs–outputs in Coq and verify them with respect to specifications by use cases. By writing programs in Coq, we gain the use of a type-safe language, with effect-free and terminating functions. We can precisely specify the behavior of our programs thanks to dependent types and the ability to prove arbitrary formal properties in the Coq language. The language of concurrent computations is a form of free monad able to describe computations with concurrency, inputs–outputs and optionally non-termination. Because this is a description and not an implementation of the effects, we do not need to add new axioms to Coq. With the method of specification by use cases, we can describe what we expect a program to do as a scenario of interactions between the environment (the user and the operating system) and the program itself. A scenario is proven correct if it is well-typed. Using the proof mode of Coq, we can write scenarios interactively with a symbolic debugger, showing the current state of a program while executing a scenario.

We provide an implementation of concurrent computations with Coq.io[1]. This library provides a set of pre-defined inputs–outputs to interact with the system (file, socket and console operations). We use the extraction mechanism of Coq to compile computations to OCaml, using the Lwt library to implement the concurrency primitives. We used the library Coq.io to write and verify the

[1] The library Coq.io, to write and verify concurrent and interactive programs in Coq, is available on coq.io.

program generating the webpages of coq.io/opam/, a website listing all the Coq packages available with the OPAM package manager. This program runs every hour to maintain the list of packages up to date.

9.1.2 Blocking computations

With *blocking computations* (chapter 8), we study a trace-based step-by-step operational semantics of the concurrent computations. We show that blocking computations are equivalent to a simpler *choose language* relying on the only operator `Choose`, which non-deterministically executes one of two computations. We give a semantics to what is a blocking interaction. For example, with a lock mechanism, we cannot call a *lock* immediately after an other *lock* and need to wait for an *unlock*. We use these semantics to implement and prove correct a checker of deadlock freedom.

9.1.3 Other forms of computations

We have studied other preliminary forms of computations. With *interactive computations* (chapter 6), we give a simplified form of concurrent computations in the special case of sequential inputs–outputs. This provides a first approach to the method of specification by use cases.

With *asynchronous computations* (chapter 5), we focus on the implementation aspect of asynchronous inputs–outputs. We interpret event handlers using an event loop, and secure the communications between the program and the operating system through a pipe and a proxy. We implement a small web server which we used to host our website.

We present *breakable computations* (chapter 4) as a monad combining the state and exception effect, together with a pause primitive. Thanks to this pause primitive, we can make explicit the evaluation steps of a computation and write a scheduler for cooperative concurrency.

9.1.4 CoqOfOCaml

The compiler CoqOfOCaml (chapter 3) translates into Coq programs written in a subset of OCaml. This subset includes polymorphic functions, records, sum-types, pattern matching, and modules. This does not include functors. We support global references, global exceptions, non-termination and basic inputs–outputs. The compiler CoqOfOCaml runs an effect inference to automatically annotate and propagate the use of effects (we do not support parametric effects). We generate an equivalent Coq code where effects are encoded into a monad. We use an operator to combine effects and make sure that functions only declare the effects they may use.

Using CoqOfOCaml, we were able to import existing OCaml modules such as `List`, `Set` and `Map`. We proved properties about the functions of these modules, and validated that the generated Coq code was still readable. Our aim is to have a compiler supporting a large subset of OCaml to import existing

programs to the relatively less used Coq language. Unfortunately, we believe that not supporting functors seriously restricts the usage of CoqOfOCaml for now, since functors are present in most of the OCaml programs we have tested.

9.1.5 Cybele

The plugin Cybele (chapter 2) helps to write proofs by reflection with effects in Coq. Proofs by reflection are proofs made with a decision procedure written and proven correct in Coq. Since we prefer decision procedures to be efficient, this was natural to try to add side-effects to Coq programs. With the idea of *simulable monads*, we can pre-run a decision procedure in OCaml to quickly test if it is successful, pre-compute some complex results and generate a *prophecy* which we import back to Coq to run the procedure with the Coq interpreter. The prophecy helps to remove some effects such as non-termination or non-determinism. With the exception effect, we can add runtime checks to simplify the formal verification of decision procedures. We use a single monad parametrized by the signature of the state to represent all the effects. We implemented some computationally intensive decision procedure in Cybele to show its applicability.

This was our first contact with effectful programming in Coq and motivated the rest of this thesis.

9.2 Future work

We present here future projects which we would find interesting.

9.2.1 CoqOfOCaml

We would like to extend CoqOfOCaml to support *functors* which are present in most OCaml programs we wanted to import. This would require some work to properly handle the naming of variables. We would then import the OCaml functors to Coq functors to respect the style of the original program, or to inlined modules for flexibility. To support a larger part of the OCaml language, we could also plug CoqOfOCaml to lower-level forms of OCaml generated by the OCaml compiler, like the byte-code OCaml. Importing low-level forms of OCaml may be at the expense of the readability of the generated Coq code.

The inference of effects is a source of complexity as it requires dedicated work to support each new OCaml construct, while being too restricted for now. For example, we do not support functions with polymorphic effects like `List.iter`. We could either improve the effect inference taking inspiration from systems such as Koka [38] or simply remove it. Without effect inference, we would generate invalid Coq programs in case of effects, but at least do the syntactic transformation from OCaml to Coq for the user.

Finally, an other axis of improvement would be to support the import of OCaml programs written using the Lwt library to the concurrent computations in Coq with the Coq.io library. This would allow the support of a whole new class of OCaml programs.

9.2.2 Use cases

We defined and studied the notion of *use cases* on some small examples. It could be interesting to study further the theory of use cases. For example, we would like to define some notion of completeness or inclusion between scenarios. We also mostly concentrated on small programs. Larger programs could have new challenges in term of composition of effects and specifications for example. Finally, our use cases are always about one program interacting with its environment, while some systems are composed of many programs communicating together. We would like to extend the study of use cases to sets of programs communicating together.

9.2.3 Blocking computations

We found the idea of having the `Choose` primitive as the only primitive to represent concurrent programs interesting in term of semantics. We would like to investigate further what we can do in this domain, and which kind of static code analyser we can write and verify in Coq. We would also like to study the use of the *choose language* as a backend to compile and implement concurrent primitives, as a replacement of the handlers and event-loop implementations.

9.2.4 Certified extraction

Finally, we did not address the validation of the extraction chain from Coq to OCaml, including the custom extraction of the effectful primitives to equivalent effectful constructs in OCaml. We can cite the Œuf[2] project or the CertiCoq project[3] which both aim to provide a certified extraction to Coq, using the certified C compiler CompCert as a backend. This would be interesting to see how to certify the extraction of effectful primitives using such certified extraction chains.

[2] The Œuf project is available online at oeuf.uwplse.org/.
[3] The CertiCoq project is available on www.cs.princeton.edu/ appel/certicoq/.

Bibliography

[1] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending coq with imperative features and its application to SAT verification. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2010. doi:10.1007/978-3-642-14052-5_8.

[2] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998. doi:10.1017/CBO9781139172752.

[3] Jan Olaf Blech and Benjamin Grégoire. Certifying compilers using higher-order theorem provers as certificate checkers. *Formal Methods in System Design*, 38(1):33–61, 2011. doi:10.1007/s10703-010-0108-7.

[4] Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005. doi:10.1017/S0960129505004822.

[5] Ana Bove and Venanzio Capretta. Computation by prophecy. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages 70–83. Springer, 2007. doi:10.1007/978-3-540-73228-0_7.

[6] Edwin Brady. IDRIS —: systems programming meets full dependent types. In Ranjit Jhala and Wouter Swierstra, editors, *Proceedings of the 5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011, Austin, TX, USA, January 29, 2011*, pages 43–54. ACM, 2011. doi:10.1145/1929529.1929536.

[7] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013. doi:10.1017/S095679681300018X.

[8] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13,*

Boston, MA, USA - September 25 - 27, 2013, pages 133–144. ACM, 2013. doi:10.1145/2500365.2500581.

[9] Edwin Brady. Resource-dependent algebraic effects. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*, volume 8843 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2014. doi:10.1007/978-3-319-14675-1_2.

[10] Edwin Brady and Kevin Hammond. Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. *Fundamenta Informaticae*, 102(2):145–176, 2010. doi:10.3233/FI-2010-303.

[11] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 418–430. ACM, 2011. doi:10.1145/2034773.2034828.

[12] Adam Chlipala. From network interface to multithreaded web applications: A case study in modular program verification. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 609–622, New York, NY, USA, 2015. ACM. doi:10.1145/2676726.2677003.

[13] Adam Chlipala. Ur/web: A simple model for programming the web. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 153–165. ACM, 2015. doi:10.1145/2676726.2677004.

[14] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 34(4):839–864, 1933. doi:10.2307/1968702.

[15] Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999. doi:10.1017/S0956796899003342.

[16] Guillaume Claret, Lourdes Del Carmen González-Huesca, Yann Régis-Gianas, and Beta Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. In *ITP*, volume 7998 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2013. doi:10.1007/978-3-642-39634-2.

[17] Guillaume Claret and Yann Régis-Gianas. Mechanical verification of interactive programs specified by use cases. In Stefania Gnesi and Nico Plat, editors, *3rd IEEE/ACM FME Workshop on Formal Methods in Software Engineering, FormaliSE 2015, Florence, Italy, May 18, 2015*, pages 61–67. IEEE Computer Society, 2015. doi:10.1109/FormaliSE.2015.17.

[18] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.

[19] Pierre Corbineau. Autour de la clôture de congruence avec coq. Master's thesis, Université Paris 7, 2001.

[20] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In Richard A. DeMillo, editor, *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, pages 207–212. ACM Press, 1982. doi:10.1145/582153.582176.

[21] David Delahaye. A tactic language for the system coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000. doi:10.1007/3-540-44404-1_7.

[22] Archibald S. Elliott. A concurrency system for Idris and Erlang. Bsc dissertation, School of Computer Science, University of St Andrews, April 2015. URL: http://lenary.co.uk/publications/bsc_dissertation.

[23] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013. doi:10.1007/978-3-642-37036-6_8.

[24] Simon Fowler and Edwin Brady. Dependent types for safe and secure web programming. In Rinus Plasmeijer, editor, *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages, Nijmegen, The Netherlands, August 28-30, 2013*, page 49. ACM, 2013. doi:10.1145/2620678.2620683.

[25] Stéphane Glondu. Extraction certifiée dans coq-en-coq. In Alan Schmitt, editor, *JFLA 2009, Vingtièmes Journées Francophones des Langages Applicatifs, Saint Quentin sur Isère, France, January 31 - February 3, 2009. Proceedings*, volume 7.2 of *Studia Informatica Universalis*, pages 383–410, 2009.

[26] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007. doi:10.1007/978-3-540-87827-8_28.

[27] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 163–175. ACM, 2011. doi:10.1145/2034773.2034798.

[28] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979. doi:10.1007/3-540-09724-4.

[29] Benjamin Grégoire, Loïc Pottier, and Laurent Théry. Proof certificates for algebra and their application to automatic geometry theorem proving. In Thomas Sturm and Christoph Zengler, editors, *Automated Deduction in Geometry - 7th International Workshop, ADG 2008, Shanghai, China, September 22-24, 2008. Revised Papers*, volume 6301 of *Lecture Notes in Computer Science*, pages 42–59. Springer, 2008. doi:10.1007/978-3-642-21046-4_3.

[30] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.

[31] Martin Hentschel, Richard Bubel, and Reiner Hähnle. Symbolic execution debugger (SED). In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*, pages 255–262. Springer, 2014. doi:10.1007/978-3-319-11164-3_21.

[32] Charles A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.

[33] William A. Howard. The Formulæ-as-types Notion of Constructions. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculs and Formalism*, pages 479–490. Academic Press, 1980.

[34] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-oriented software engineering - a use case driven approach*. Addison-Wesley, 1992.

[35] Daniel W. H. James and Ralf Hinze. A reflection-based proof tactic for lattices in coq. In Zoltán Horváth, Viktória Zsók, Peter Achten, and Pieter W. M. Koopman, editors, *Proceedings of the Tenth Symposium on Trends in Functional Programming, TFP 2009, Komárno, Slovakia, June 2-4, 2009.*, volume 10 of *Trends in Functional Programming*, pages 97–112. Intellect, 2009.

[36] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 247–259. ACM, 2015. doi:10.1145/2676726.2676966.

[37] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. doi:10.1145/360248.360252.

[38] Daan Leijen. Koka: Programming with row polymorphic effect types. In Paul Levy and Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014.*, volume 153 of *EPTCS*, pages 100–126, 2014. doi:10.4204/EPTCS.153.8.

[39] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

[40] Pierre Letouzey. Extraction in coq: An overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008. doi:10.1007/978-3-540-69407-6_39.

[41] Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In Ron K. Cytron and Peter Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 333–343. ACM Press, 1995. doi:10.1145/199448.199528.

[42] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. Trace-based verification of imperative programs with I/O. *Journal of Symbolic Computation*, 46(2):95–118, 2011. doi:10.1016/j.jsc.2010.08.004.

[43] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992. doi:10.1007/978-1-4612-0931-7.

[44] Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990. doi:10.1007/978-0-262-63132-7.

[45] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23. IEEE Computer Society, 1989. doi:10.1109/LICS.1989.39155.

[46] Aleksandar Nanevski, Paul Govereau, and Greg Morrisett. Towards type-theoretic semantics for transactional concurrency. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 79–90. ACM, 2009. doi:10.1145/1481861.1481872.

[47] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 229–240. ACM, 2008. doi:10.1145/1411204.1411237.

[48] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015. doi:10.1145/2699417.

[49] Simon Peyton-Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In C. A. R. Hoare, M. Broy, and R. Steinbrueggen, editors, *Engineering Theories of Software Construction*, NATO ASI Series, pages 47–96. IOS Press, 2001. Marktoberdorf Summer School 2000.

[50] Simon Peyton-Jones. Beautiful concurrency. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly Media, Inc., January 2007. URL: https://www.microsoft.com/en-us/research/publication/beautiful-concurrency/.

[51] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009. doi:10.1007/978-3-642-00590-9_7.

[52] Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *SIGPLAN Notices*, 38(2):57–64, 2003. doi:10.1145/772970.772977.

[53] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.

[54] Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Automating formal proofs for reactive systems. In Michael

F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 452–462. ACM, 2014. doi:10.1145/2594291.2594338.

[55] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 77–87. ACM, 2015. doi:10.1145/2737924.2737964.

[56] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 333–358. Springer, 2015. doi:10.1007/978-3-662-46669-8_14.

[57] Stéphane S. Somé. Formalization of textual use cases based on petri nets. *International Journal of Software Engineering and Knowledge Engineering*, 20(5):695–737, 2010. doi:10.1142/S0218194010004931.

[58] Matthieu Sozeau. Subset coercions in coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2006. doi:10.1007/978-3-540-74464-1_16.

[59] Matthieu Sozeau. Program-ing finger trees in coq. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 13–24. ACM, 2007. doi:10.1145/1291220.1291156.

[60] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[61] Rob J. van Glabbeek and Gordon D. Plotkin. On CSP and the algebraic theory of effects. In A. W. Roscoe, Clifford B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C. A. R. Hoare.*, pages 333–369. Springer, 2010. doi:10.1007/978-1-84882-912-1_15.

[62] Jérôme Vouillon. Lwt: a cooperative thread library. In Eijiro Sumii, editor, *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada,*

September 21, 2008, pages 3–12. ACM, 2008. doi:10.1145/1411304.1411307.

[63] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992. doi:10.1017/S0960129500001560.

[64] Philip M. Whitman. Free lattices. *Annals of Mathematics*, 42(1):325–330, 1941. doi:10.2307/1969001.

[65] Beta Ziliani and Matthieu Sozeau. A unification algorithm for coq featuring universe polymorphism and overloading. In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 179–191. ACM, 2015. doi:10.1145/2784731.2784751.