A theory of monads parameterized by effects

Jean-Christophe Filliâtre

LRI, Université Paris Sud
filliatr@lri.fr, www.lri.fr/~filliatr

November 26, 1999

Topics covered: semantics, monads, effects.

Abstract

Monads were introduced in computer science to express the semantics of programs with computational effects, while type and effect inference was introduced to mark out those effects. In this article, we propose a combination of the notions of effects and monads, where the monadic operators are parameterized by effects. We establish some relationships between those generalized monads and the classical ones. Then we use a generalized monad to translate imperative programs into purely functional ones. We establish the correctness of that translation. This work has been put into practice in the Coq proof assistant to establish the correctness of imperative programs.

1 Introduction

It is well known that purely functional programs are easier to reason about than imperative ones, and one of the first advice of Kernighan and Pike's *Practice of Programming* [8] is precisely "Be careful with side effects". When reasoning about a program, we expect expressions to be substitutable and we want to manipulate variables as mathematical ones, and this is only possible with purely functional programs. Thus, when we faced the problem of certifying imperative programs in the Coq proof assistant [1, 3, 4], it was natural to look for a functional interpretation of those programs.

Monads were introduced in computer science to express the semantics of programs with computational effects, first by Moggi [10] and then by Wadler [13]. A monad introduces a type operator μ, where $\mu\ \tau$ represents computations returning values of type τ. The type $\mu\ \tau$ may be seen as an abstract datatype, which is manipulated through two primitive operations. Operation $\mathsf{unit} : \tau \rightarrow \mu\ \tau$ creates an elementary computation from a value, and operation $\mathsf{star} : \mu\ \tau \rightarrow (\tau \rightarrow \mu\ \tau') \rightarrow \mu\ \tau'$ composes two computations, binding the result of the first one into the second one. The strength of monads is to be defined independently of a particular notion of effect. Of course, for a given effect, you may have other primitive monadic operations; in the case of a monad to handle a store, for instance, you will have operations to create a new reference, and to get or to set its value. Monads are used in purely functional languages, such as Haskell [2], where they can be seen as "internalized denotational semantics".

Unfortunately, monads appear to be useful from an operational point of view, but not precise enough when reasoning about programs. Indeed, if a program which modifies a single variable x is interpreted as a function over the whole store, then you will have to express in the specification the properties of that whole store; and you will be quickly lost into properties expressing that most of the variables are unmodified. On the other hand, if you decide to use a monad where the store only contains the variables declared so far, then you will not be able to reuse that interpretation — and the corresponding proof of correctness — in a context containing additional variables. In other words, the method will not be incremental.

The ideal solution would be to interpret an imperative program as a function taking the necessary values as arguments and returning all the resulting values, as one does when writing a purely functional program instead of an imperative one. Then it is easy to reason about such interpretations, and they are easy to compose, even when the context grows. It implies to be able to determine the variables possibly modified by a computation. Such an inference has been a subject of research for many years now, and the most widely known work is surely the type and effect inference of Talpin and Jouvelot [11, 12].

To exploit the benefits of an effect inference, we have to refine the notion of monad. We propose a notion of generalized monad, where the monadic operators are parameterized by effects. The monadic operator μ is now indexed by an effect, $\mu_\epsilon\,\tau$ being the type of computations of effect ϵ and type τ. The operation unit becomes a coercion operation $\mathsf{unit}_{\epsilon_1,\epsilon_2}$, which transforms a computation of effect ϵ_1 into a computation of effect ϵ_2, as soon as ϵ_2 is greater than ϵ_1. And star becomes an operation $\mathsf{star}_{\epsilon_1,\epsilon_2}$ to compose two computations of respective effects ϵ_1 and ϵ_2. With its single monadic operator, a classical monad may be seen as associated to a trivial notion of effect: there is an effect or there is not.

Outline. This paper is organized as follows. Section 2 is devoted to the presentation of generalized monads. We show their relationships with the classical ones. We also introduce a type system with effects, and we define a canonical way to apply the monadic operator to arbitrary types. Section 3 defines a generalized monad for references. Then it is applied to the functional translation of imperative programs, through the usual monadic call-by-value translation. We show the correctness of that translation. Section 4 is a brief conclusion, in which we discuss the differences with a similar work of Wadler recently published in [14].

2 Generalized monads

This section introduces the notion of generalized monads, and some of their properties.

2.1 Definitions

First, let us recall the classical definition of a monad.

Definition 1 (monad, [10, 13]) *A monad is defined by a type operator μ, and two operations*

$$\begin{array}{lcl} \mathsf{unit} & : & \tau \to \mu\ \tau \\ \mathsf{star} & : & \mu\ \tau \to (\tau \to \mu\ \tau') \to \mu\ \tau' \end{array}$$

which have to satisfy the following three axioms:

$$\begin{array}{ll} (M_1) & (\mathsf{star}\ (\mathsf{unit}\ x)\ f) = (f\ x) \\ (M_2) & (\mathsf{star}\ m\ \lambda x.(\mathsf{unit}\ x)) = m \\ (M_3) & (\mathsf{star}\ m\ \lambda x.(\mathsf{star}\ (g\ x)\ f)) = (\mathsf{star}\ (\mathsf{star}\ m\ g)\ f) \end{array}$$

Then, we have to set an abstract notion of effect. Obviously, we need a particular element to represent the absence of effect, say $\bot$. We will also need an operation to make the union of two effects, when composing computations; let us write $\sqcup$ that operation. We expect it to be commutative, associative, idempotent and to have $\bot$ as a neutral element. Finally, we have to express that an effect is more precise than another one, and it induces a partial order relation $\sqsubseteq$ over effects, which could be defined by $x \sqsubseteq y$ iff $y = x \sqcup y$. This is exactly the structure of a semilattice.

Definition 2 (effect) *An effect is a semilattice $(E, \bot, \sqcup, \sqsubseteq)$.*

The simplest effect is the lattice of booleans $\mathcal{B}$, whose elements are $\{\bot, \top\}$.

Given an effect $\mathcal{E}$, a generalized monad will look like a classical one, but with three operators μ , unit and star now indexed over effects. Moreover, the operations of a generalized monad will have to satisfy axioms similar to (M_1)–(M_3). The formal definition is given below.

Definition 3 (generalized monad) *A* generalized monad *associated to an effect $\mathcal{E} = (E, \bot, \sqcup, \sqsubseteq)$ is given by a type operator μ_ϵ indexed over E, and two families of operations, $\mathsf{unit}_{\epsilon_1,\epsilon_2}$ indexed over $\{(\epsilon_1, \epsilon_2) \in E^2 \mid \epsilon_1 \sqsubseteq \epsilon_2\}$, and $\mathsf{star}_{\epsilon_1,\epsilon_2}$ indexed over E^2, whose types are:*

$$\begin{array}{lcl} \mathsf{unit}_{\epsilon_1,\epsilon_2} & : & \mu_{\epsilon_1}\,\tau \to \mu_{\epsilon_2}\,\tau \\ \mathsf{star}_{\epsilon_1,\epsilon_2} & : & \mu_{\epsilon_1}\,\tau \to (\tau \to \mu_{\epsilon_2}\,\tau') \to \mu_{\epsilon_1 \sqcup \epsilon_2}\,\tau' \end{array}$$

and which satisfy the following six identities:

$$\begin{array}{ll}
(G_1) & \mu_\perp\,\tau = \tau \\
(G_2) & (\mathsf{unit}_{\epsilon,\epsilon}\ x) = x \\
(G_3) & (\mathsf{star}_{\perp,\epsilon}\ x\ f) = (f\ x) \\
(G_4) & (\mathsf{star}_{\epsilon_1,\epsilon_2}\ m\ \lambda x.(\mathsf{unit}_{\perp,\epsilon_2}\ x)) = (\mathsf{unit}_{\epsilon_1,\epsilon_1\sqcup\epsilon_2}\ m) \\
(G_5) & (\mathsf{star}_{\epsilon_1,\epsilon_2\sqcup\epsilon_3}\ m\ \lambda x.(\mathsf{star}_{\epsilon_2,\epsilon_3}\ (g\ x)\ f)) = (\mathsf{star}_{\epsilon_1\sqcup\epsilon_2,\epsilon_3}\ (\mathsf{star}_{\epsilon_1,\epsilon_2}\ m\ g)\ f) \\
(G_6) & (\mathsf{star}_{\epsilon_2,\epsilon_3}\ (\mathsf{unit}_{\epsilon_1,\epsilon_2}\ x)\ f) = (\mathsf{unit}_{\epsilon_1\sqcup\epsilon_3,\epsilon_2\sqcup\epsilon_3}\ (\mathsf{star}_{\epsilon_1,\epsilon_3}\ x\ f)) \qquad \textit{given } \epsilon_1 \sqsubseteq \epsilon_2
\end{array}$$

The types of the operations unit and star are self-explainable. The axiom (G_1) expresses that the type of a computation with a null effect $\perp$ is directly interpreted by itself. The axiom (G_2) states that the coercion operation unit is the identity when no coercion is needed. The axiom (G_3) is similar to the axiom (M_1). The axiom (G_4) generalizes the axiom (M_2) for any function coercing the result of a computation in any greater effect. Notice that it implies that $(\mathsf{star}_{\epsilon,\epsilon}\ m\ \lambda x.x) = m$, when $\epsilon_2 = \perp$ and with the use of axiom (G_2) twice, which is exactly the axiom (M_2). The axiom (G_5) is exactly the associative law stated by axiom (M_3). The axiom (G_6) states a property of commutation between unit and star: we can first coerce a computation x from effect ϵ_1 to effect ϵ_2 and then apply a function f to its value, or we can apply f to the value of x and then coerce the result from $\epsilon_1 \sqcup \epsilon_2$ to $\epsilon_1 \sqcup \epsilon_3$, equivalently.

2.2 Relationship with classical monads

We expect the classical monads to be a particular case of generalized monads, and it is indeed the case. Let $M_0 = (\mu_0, \mathsf{unit}_0, \mathsf{star}_0)$ be a classical monad. We can define from M_0 a generalized monad associated to the boolean effects $\mathcal{B}$, where the element $\perp$ will represent the absence of effect, and the element $\top$ the presence of effect. We define the type operator μ by:

$$\mu_\perp = id \qquad \text{and} \qquad \mu_\top = \mu_0$$

We define the operation unit by:

$$\mathsf{unit}_{\perp,\perp} = \mathsf{unit}_{\top,\top} = id \qquad \text{and} \qquad \mathsf{unit}_{\perp,\top} = \mathsf{unit}_0$$

We define the operation star by:

$$\begin{array}{lclcl}
\mathsf{star}_{\perp,\perp} & : & \tau \to (\tau \to \tau') \to \tau' & = & \lambda x.\lambda f.(f\ x) \\
\mathsf{star}_{\perp,\top} & : & \tau \to (\tau \to \mu_0\,\tau') \to \mu_0\,\tau' & = & \lambda x.\lambda f.(f\ x) \\
\mathsf{star}_{\top,\perp} & : & \mu_0\,\tau \to (\tau \to \tau') \to \mu_0\,\tau' & = & \lambda x.\lambda f.(\mathsf{star}_0\ x\ \lambda v.(\mathsf{unit}_0\ (f\ v)) \\
\mathsf{star}_{\top,\top} & : & \mu_0\,\tau \to (\tau \to \mu_0\,\tau') \to \mu_0\,\tau' & = & \mathsf{star}_0
\end{array}$$

Proposition 1 *The operations* $(\mu, \mathsf{unit}, \mathsf{star})$ *defined above from a classical monad* $(\mu_0, \mathsf{unit}_0, \mathsf{star}_0)$ *form a generalized monad associated to the lattice of booleans i.e. satisfy the identities* (G_1)*–*(G_6)*.*

Conversely, given a generalized monad $G = (\mu, \mathsf{unit}, \mathsf{star})$ associated to an effect $\mathcal{E} = (E, \perp, \sqcup, \sqsubseteq)$, and given a particular element $\epsilon_0 \in E$, we can define the following operators:

$$\begin{array}{lcl}
\mu_0 & = & \mu_{\epsilon_0} \\
\mathsf{unit}_0 & = & \mathsf{unit}_{\perp,\epsilon_0} \\
\mathsf{star}_0 & = & \mathsf{star}_{\epsilon_0,\epsilon_0}
\end{array}$$

Proposition 2 *The operations* $(\mu_0, \mathsf{unit}, \mathsf{star}_0)$ *defined above form a classical monad i.e. satisfy the identities* (M_1)*–*(M_3)*.*

The two propositions above are proved straightforwardly, by a simple equational reasoning. The proofs are given in [4], pages 43–45.

2.3 Type systems with effects

Once a notion of effect is given, it is natural to define a type system with effects, and to do type and effect inference at the same time, as it is done for instance by Talpin and Jouvelot for references [11] or by others for exceptions [6, 7]. From now on, we consider a call-by-value semantics, to the end of the paper. Then there is a generic way to incorporate effects within the functional part of a type system, but also a canonical way to extend some operations of a generalized monad to complex types.

To present a type system with effects, it is convenient to distinguish between types of values τ and types of expressions κ, where an expression produces an effect and computes a value. Given base types ι, such a type system looks like:

$$\begin{array}{rcl} \tau & ::= & \iota \mid \tau \to \kappa \\ \kappa & ::= & (\tau, \epsilon) \end{array}$$

The type of a function has the shape $\tau \to (\tau', \epsilon)$, where τ is the domain, τ' the range, and ϵ the effect resulting from the application of the function. Such a type is often written $\tau \xrightarrow{\epsilon} \tau'$, and we will use that notation in the following. The presentation as $\tau \to \kappa$ better illustrates the fact that a function takes a value and returns a computation.

There is a canonical way to extend the partial order relation $\sqsubseteq$ over effects to the types τ and κ. It is the least reflexive and transitive relation such that:

$$\begin{array}{lll} \tau_1 \to \kappa_1 \sqsubseteq \tau_2 \to \kappa_2 & \text{iff} & \tau_2 \sqsubseteq \tau_1 \ \wedge \ \kappa_1 \sqsubseteq \kappa_2 \\ (\tau_1, \epsilon_1) \sqsubseteq (\tau_2, \epsilon_2) & \text{iff} & \tau_1 \sqsubseteq \tau_2 \ \wedge \ \epsilon_1 \sqsubseteq \epsilon_2 \end{array}$$

There is also a canonical way to extend the type operator μ of a generalized monad to the types τ and κ, in the following way:

$$\begin{array}{lcl} \mu\ \iota & \stackrel{\text{def}}{=} & \iota \\ \mu\ (\tau \to \kappa) & \stackrel{\text{def}}{=} & \mu\ \tau \to \mu\ \kappa \\ \mu\ (\tau, \epsilon) & \stackrel{\text{def}}{=} & \mu_\epsilon\ (\mu\ \tau) \end{array} \tag{1}$$

Then the coercion operation unit of a generalized monad may be lifted from the level of effects to the level of types, giving an operation $\mathsf{unit}_{\tau_1,\tau_2} : \mu(\tau_1) \to \mu(\tau_2)$ for $\tau_1 \sqsubseteq \tau_2$ and an operation $\mathsf{unit}_{\kappa_1,\kappa_2} : \mu(\kappa_1) \to \mu(\kappa_2)$ for $\kappa_1 \sqsubseteq \kappa_2$. Those operations are defined recursively by:

$$\begin{array}{lcl} \mathsf{unit}_{\tau,\tau} & \stackrel{\text{def}}{=} & \lambda x.x \\ \mathsf{unit}_{\tau_1 \to \kappa_1, \tau_2 \to \kappa_2} & \stackrel{\text{def}}{=} & \lambda f.\lambda x.(\mathsf{unit}_{\kappa_1,\kappa_2}\ (f\ (\mathsf{unit}_{\tau_2,\tau_1}\ x))) \\ \mathsf{unit}_{(\tau_1,\epsilon_1),(\tau_2,\epsilon_2)} & \stackrel{\text{def}}{=} & \lambda x.(\mathsf{star}_{\epsilon_1,\epsilon_2}\ x\ \lambda v.(\mathsf{unit}_{\perp,\epsilon_2}\ (\mathsf{unit}_{\tau_1,\tau_2}\ v))) \end{array}$$

Such operations express that any object of type τ_1 (resp. κ_1) may be considered as an object of type τ_2 (resp. κ_2) as soon as $\tau_1 \sqsubseteq \tau_2$ (resp. $\kappa_1 \sqsubseteq \kappa_2$), i.e. that an object may always be considered as having more effects than it really has.

3 Functional translation of imperative programs

This section defines a purely functional interpretation for an imperative programming language, which is rather simple without being too naive. That translation uses a generalized monad for references, which tries to give a functional expression as close as possible as a hand-written functional program from the imperative program. Such a monadic translation is not new — it is defined by Moggi in [9] and we only add the effects component — but contrary to other works, we make the monad explicit to get a purely functional translation of imperative programs.

3.1 A simple imperative language and its effect typing

We consider a simple imperative programming language with higher-order functions and references, whose expressions are defined by the following grammar:

$$e \ ::= \ c \mid x \mid \lambda x.e \mid (e\ e) \mid \mathsf{rec}\ x\ x = e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{ref}\ e \mid \ !x \mid x\ := \ e$$

c stands for constants, and we assume that we have at least a type bool of booleans and a type unit with a single value void. The expression $\mathsf{rec}\ f\ x = e$ defines a recursive function f, with f and x bound in e. A new reference is created by $\mathsf{ref}\ e$, dereference by $!x$ and assigned with $x\ :=\ e$. Notice that operations of dereference and assignment only operate on variables i.e. named references, and not on anonymous expressions. We have local references, using the $\mathsf{let}\ \mathsf{in}$ construction, and procedures, as functions taking references as arguments. Such a programming language may be seen as a ML kernel with references, or a core Pascal, if we omit higher-order features.

A type system for the above language would contain base types ι, function types and a type construction $\mathsf{ref}\ \tau$ for references containing values of types τ, that is:

$$\tau \ ::= \ \iota \mid \tau \rightarrow \tau \mid \mathsf{ref}\ \tau$$

but we prefer to give directly a type system with effects. An effect ϵ is here composed of two sets of variables, a first set ρ containing the variables the program may access, and a second one ω containing the variables it possibly modifies. We impose that $\omega \subseteq \rho$. The empty effect $(\emptyset, \emptyset)$ is still written $\bot$. The sup of two effects is defined by $(\rho_1, \omega_1) \sqcup (\rho_2, \omega_2) = (\rho_1 \cup \rho_2, \omega_1 \cup \omega_2)$. The order relation over effects is defined by $(\rho_1, \omega_1) \sqsubseteq (\rho_2, \omega_2)$ iff $\rho_1 \subseteq \rho_2 \wedge \omega_1 \subseteq \omega_2$. The type system with effects is the following:

$$\begin{aligned} \tau &\ ::= \ \iota \mid (x : \tau) \rightarrow \kappa \mid \mathsf{ref}\ \tau \\ \kappa &\ ::= \ (\tau, \epsilon) \end{aligned}$$

The argument of a function of type $\tau \rightarrow \kappa$ is given a name when it is a reference, since it may appear in effects contained in κ, where it is then bound. A typing environment Γ is a list of bindings of a type of value τ to a variable. The typing judgment associates, in a given environment Γ, a type of expression κ to an expression e, and is written $\Gamma \vdash e : \kappa$. The typing rules are given in Figure 1.

Exclusion of aliases. Since we want to determine for each sub-expression its sets of variables possibly accessed or modified, we should immediately face the well-known problem of *aliases*. As you may have noticed, they have been subtly excluded by some restrictions on the syntax and the typing rules. Aliases resulting from a let binding are excluded by the typing rule (VAR), which does not allow to manipulate a reference x anywhere else than in $!x$, $x\ :=\ e$ and $(e\ x)$. And aliases resulting from an application of a function twice to the same reference are excluded by the typing rule (APPREF), which checks that a reference passed to a function does not already appear in the effect resulting from the application, even after further applications. Notice that is does not exclude at all the call-by-variable.

The exclusion of aliases is the price to pay when we want to reason easily about programs; dealing with aliases would make specifications and proofs of correctness really more complicated — even if it is possible, using for instance a static analysis based on regions. However, the programming language we chose appeared to be powerful enough in practice to write (and prove correct) quite complex algorithms.

3.2 Definition of the translation

The target language of our translation is a typed-lambda calculus, with the same base types ι than the source language, and a fix-point operator still written rec. We choose for the target language the same call-by-value semantics than the programming language, where arguments are evaluated before functions, and where no reduction is done under abstractions. (It is important, since we can write non-terminating programs in the target language.) We also assume that the target language comes with a primitive notion of records, whose types are written $\{x_1 : \tau_1; \ldots; x_n : \tau_n\}$ and elements $\{x_1 = v_1; \ldots; x_n = v_n\}$. If s is such a record, then $s.x$ stands for the value of its field x. We also assume that it is possible to abstract a term with respect

$$\frac{\mathit{Type}(c) = \tau}{\Gamma \vdash c : (\tau, \bot)}(\mathsf{CONST}) \qquad \frac{x : \tau \in \Gamma \qquad \tau \neq \mathsf{ref}\ \_}{\Gamma \vdash x : (\tau, \bot)}(\mathsf{VAR}) \qquad \frac{\Gamma, x : \tau \vdash e : \kappa}{\Gamma \vdash \lambda x.e : ((x : \tau) \to \kappa, \bot)}(\mathsf{FUN})$$

$$\frac{\Gamma \vdash e_2 : (\tau_1 \xrightarrow{\epsilon} \tau_2, \epsilon_2) \qquad \Gamma \vdash e_1 : (\tau_1, \epsilon_1) \qquad \tau_1 \neq \mathsf{ref}\ \_}{\Gamma \vdash (e_2\ e_1) : (\tau_2, \epsilon_2 \sqcup \epsilon_1 \sqcup \epsilon)}(\mathsf{APP})$$

$$\frac{\Gamma \vdash e : ((x : \mathsf{ref}\ \tau_1) \xrightarrow{\epsilon} \tau_2, \epsilon_2) \qquad r : \mathsf{ref}\ \tau_1 \in \Gamma \qquad r \notin (\tau_2, \epsilon)}{\Gamma \vdash (e\ r) : (\tau_2[x \leftarrow r], \epsilon_2 \sqcup \epsilon[x \leftarrow r])}(\mathsf{APPREF})$$

$$\frac{\Gamma, f : \tau \to \kappa, x : \tau \vdash e : \kappa}{\Gamma \vdash \mathsf{rec}\ f\ x = e : (\tau \to \kappa, \bot)}(\mathsf{REC})$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, \epsilon_1) \qquad \tau_1 \neq \mathsf{ref}\ \_ \qquad \Gamma, x : \tau_1 \vdash e_2 : (\tau_2, \epsilon_2)}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : (\tau_2, \epsilon_1 \sqcup \epsilon_2)}(\mathsf{LET})$$

$$\frac{\Gamma \vdash e_1 : (\mathsf{ref}\ \tau_1, \epsilon_1) \qquad \Gamma, x : \mathsf{ref}\ \tau_1 \vdash e_2 : (\tau, \epsilon) \qquad x \notin \tau}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : (\tau, \epsilon_1 \sqcup \epsilon \backslash x)}(\mathsf{LETREF})$$

$$\frac{\Gamma \vdash e_1 : (\mathsf{bool}, \epsilon_1) \qquad \Gamma \vdash e_2 : (\tau, \epsilon_2) \qquad \Gamma \vdash e_3 : (\tau, \epsilon_3)}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : (\tau, \epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon_3)}(\mathsf{COND})$$

$$\frac{\Gamma \vdash e : (\tau, \epsilon) \qquad \tau \text{ purely functional}}{\Gamma \vdash \mathsf{ref}\ e : (\mathsf{ref}\ \tau, \epsilon)}(\mathsf{REF}) \qquad \frac{x : \mathsf{ref}\ \tau \in \Gamma}{\Gamma \vdash !x : (\tau, (\{x\}, \emptyset))}(\mathsf{DEREF})$$

$$\frac{x : \mathsf{ref}\ \tau \in \Gamma \qquad \Gamma \vdash e : (\tau, (\rho, \omega))}{\Gamma \vdash x := e : (\mathsf{unit}, (\{x\} \cup \rho, \{x\} \cup \omega))}(\mathsf{ASSIGN})$$

Figure 1: Typing rules

to a label, and to apply such an abstraction to a given label, with the expected substitution. (This is not very realistic, and we explain in [4] how to manage with anonymous tuples instead; however, it simplifies the current presentation, without lost of generality.) The target language only contains two notions of reduction, which are the β-reduction and the field access reduction. We will write $\triangleright$ the union of those two reductions, and $\triangleright^\star$ its transitive closure, as usual.

We have defined the lattice of effects in the previous section. Then we have to define the associated generalized monad. If Γ is an environment containing some references $x_i : \mathsf{ref}\ \tau_i$, and if $\epsilon = (\rho, \omega)$ is an effect well-formed in that environment, say $\rho = \{x_{\phi(i)} \mid i = 1, \ldots, k\}$ and $\omega = \{x_{\psi(i)} \mid i = 1, \ldots, m\}$, then the monadic operator $\mu_\epsilon\,\tau$ is defined by:

$$\mu_{(\rho,\omega)}(\tau) \quad \stackrel{\text{def}}{=} \quad \{x_{\phi(1)} : \tau_{\phi(1)}; \ldots; x_{\phi(k)} : \tau_{\phi(k)}\} \to \{x_{\psi(1)} : \tau_{\psi(1)}; \ldots; x_{\psi(m)} : \tau_{\psi(m)}\} \times \tau$$

Stated more simply, it expresses that we interpret a program as a function taking as argument a record containing the values of the references involved in the computation, and returning another record with the possibly modified values, together with the result itself.

To define the operations unit and star, we need some operations over records. The first one is used to suppress some fields — which should be made implicit in presence of sub-typing. Formally, if s is a record containing at least the fields l_i, we define $[s]_{\{l_1,\ldots,l_n\}}$ as the record $\{l_1 = s.l_1; \ldots; l_n = s.l_n\}$. The second operation is used to "update" some record s with another one s', giving a record with all the fields of s and

$$
\begin{array}{rcl}
c^\star & = & c \\
x^\star & = & x \\
(\lambda x.e)^\star & = & \lambda x.e^\star \\
(\mathsf{rec}\ f\ x = e)^\star & = & \mathsf{rec}\ f\ x = e^\star \\
(e_2\ e_1)^\star & = & \mathsf{star}_{\epsilon_1,\epsilon_2\sqcup\epsilon}\ e_1{}^\star\ \lambda x.(\mathsf{star}_{\epsilon_2,\epsilon}\ e_2{}^\star\ \lambda f.(f\ x)) \\
(e_2\ r)^\star & = & \mathsf{star}_{\epsilon_2,\epsilon[x\leftarrow r]}\ e_2{}^\star\ \lambda f.(f\ r) \\
(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)^\star & = & \mathsf{star}_{\epsilon_1,\epsilon_2}\ e_1{}^\star\ \lambda x.e_2{}^\star \\
(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)^\star & = & \mathsf{star}_{\epsilon_1,\epsilon_2\backslash x}\ e_1{}^\star\ \lambda v.\lambda s.(\mathsf{let}\ (s',v') = (e_2{}^\star\ s \oplus \{x = v\})\ \mathsf{in}\ ([s']_{\omega_2\backslash x}, v')) \\
 & & \text{when } e_1 : \mathsf{ref}\ \tau_1 \\
(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)^\star & = & \mathsf{star}_{\epsilon_1,\epsilon_2\sqcup\epsilon_3}\ e_1{}^\star\ \lambda b.(\mathsf{if}\ b\ \mathsf{then}\ (\mathsf{unit}_{\epsilon_2,\epsilon_2\sqcup\epsilon_3}\ e_2{}^\star)\ \mathsf{else}\ (\mathsf{unit}_{\epsilon_3,\epsilon_2\sqcup\epsilon_3}\ e_3{}^\star)) \\
(\mathsf{ref}\ e)^\star & = & e^\star \\
(!x)^\star & = & \lambda s.\ s.x \\
(x := e)^\star & = & \lambda s.\mathsf{let}\ (s',v) = (e^\star\ s)\ \mathsf{in}\ (s' \oplus \{x = v\}, \mathsf{void})
\end{array}
$$

Figure 2: Translation of programs

s', and with the values of s' for all the fields appearing in s'. Formally, if $s = \{x_1 = u_1; \ldots; x_n = u_n\}$ and $s' = \{y_1 = v_1; \ldots; y_m = v_m\}$ then $s \oplus s'$ stands for the record $\{z_1 = w_1; \ldots; z_k = w_k\}$ where $\{z_i\}_{i=1\ldots k} = \{x_i\}_{i=1\ldots n} \cup \{y_i\}_{i=1\ldots m}$ and where $w_i = v_j$ if $z_i = y_j$ and $w_i = u_j$ if $z_i = x_j$ and $z_i \notin \{y_j\}$.

Then the two operations unit and star are defined in the following way:

$$\mathsf{unit}_{(\rho_1,\omega_1),(\rho_2,\omega_2)}\ m \quad \stackrel{\text{def}}{=} \quad \lambda s.\ \mathsf{let}\ (s',v) = (m\ [s]_{\rho_1})\ \mathsf{in}\ ([s \oplus s']_{\omega_2}, v)$$

and

$$
\begin{array}{lcl}
\mathsf{star}_{(\rho_1,\omega_1),(\rho_2,\omega_2)}\ m\ f & \stackrel{\text{def}}{=} & \lambda s.\ \mathsf{let}\ (s',v_1) = (m\ [s]_{\rho_1})\ \mathsf{in} \\
 & & \quad\ \mathsf{let}\ (s'',v_2) = (f\ v_1\ [s \oplus s']_{\rho_2})\ \mathsf{in} \\
 & & \quad\ ([s \oplus s' \oplus s'']_{\omega_1\cup\omega_2}, v_2)
\end{array}
$$

You can notice that the inclusion $\omega \subseteq \rho$ is really needed in the definition of unit, since the returned values which do not come from the evaluation of m are taken in the input state s.

We can now define the translation of types, written $\tau^\star$ and $\kappa^\star$, by:

$$
\begin{array}{lcll}
\iota^\star & = & \iota & \\
(\mathsf{ref}\ \tau)^\star & = & \tau & \text{since } \tau \text{ is purely functional} \\
((x:\tau) \to \kappa)^\star & = & \tau^\star \to \kappa^\star & \\
(\tau,\epsilon)^\star & = & \mu_\epsilon(\tau^\star) &
\end{array}
$$

This is exactly the canonical extension of μ to the types given by (1), with an additional case for the type constructor ref. We use here the fact that values of type $\mathsf{ref}\ \tau$ are necessarily new references to interpret them directly as the values they contain. If $x_i : \tau_i$ are the variables of a typing environment Γ which are not references, then we define the translated environment by $\Gamma^\star = x_1 : \tau_1{}^\star, \ldots, x_n : \tau_n{}^\star$.

The translation of programs is given in Figure 2. The effects involved in the translation are the ones appearing in the typing rules of Figure 1. The following property is easily established by a structural induction over expression e.

Proposition 3 *The types are preserved by the translation i.e.*

$$\text{If} \quad \Gamma \vdash e : \kappa \quad \text{then} \quad \Gamma^\star \vdash e^\star : \kappa^\star$$

$$
\begin{array}{llll}
\textit{expressions} & e & ::= & v \mid (e\ e) \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \theta.e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{ref}\ e \mid !x \mid x := e \\
\textit{values} & v & ::= & c \mid x \mid \lambda x.e \mid \mathsf{rec}\ x\ x = e \\
\textit{stores} & \theta & ::= & \{(x,v),\ldots,(x,v)\}
\end{array}
$$

$$
\begin{array}{rcllr}
(\lambda x.e\ v) & \longrightarrow & e[x \leftarrow v] & & (\beta) \\
\mathsf{let}\ x = v\ \mathsf{in}\ e & \longrightarrow & e[x \leftarrow v] & & (\textit{let}) \\
(\mathsf{rec}\ f\ x = e\ v) & \longrightarrow & (\lambda x.e[f \leftarrow \mathsf{rec}\ f\ x = e]\ v) & & (\mathsf{rec}) \\
\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \longrightarrow & e_1 & & (\textit{if-true}) \\
\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \longrightarrow & e_2 & & (\textit{if-false}) \\
(\mathsf{ref}\ v) & \longrightarrow & \{(x,v)\}.x & x \text{ fresh} & (\textit{ref}) \\
\theta.\,!x & \longrightarrow & v & (x,v) \in \theta & (\textit{deref}) \\
\theta.\,x := v & \longrightarrow & \theta \cup \{(x,v)\}.\,\mathsf{void} & x \in \textit{dom}(\theta) & (\textit{assign}) \\
\theta.\,\theta'.\,e & \longrightarrow & \theta \cup \theta'.\,e & & (\textit{merge})
\end{array}
$$

$$
E \quad ::= \quad [] \mid (E\ v) \mid (e\ E) \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid \mathsf{if}\ E\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \theta.E \mid \mathsf{ref}\ E \mid x := E
$$

$$
E[e] \longmapsto E[e'] \quad \text{iff} \quad e \longrightarrow e'
$$

Figure 3: Semantics for programs

3.3 Correctness of the translation

To justify our translation, we have to show that e and $e^\star$ compute the same values, and it implies to define first a formal semantics for the imperative programs. We chose to follow a small steps reduction semantics introduced by Wright and Felleisen in [15] to give a formal semantics to the core of SML with references. The reduction rules are summarized in Figure 3. For simplicity, we assume here that constants have only base types. $\longmapsto^\star$ will denote the reflexive transitive closure of the reduction relation $\longmapsto$.

Since semantical values may be references, we have to interpret them correctly. In the translation, we chose to interpret a new reference as its value, and therefore the interpretation of a value v in a store θ, written $\textit{val}(\theta, v)$, is defined by:

$$
\begin{array}{lcll}
\textit{val}(\theta,x) & \stackrel{\text{def}}{=} & \theta(x) & \text{if } x \in \textit{dom}(\theta) \\
\textit{val}(\theta,v) & \stackrel{\text{def}}{=} & v & \text{otherwise}
\end{array}
$$

Then we can express the semantical correctness of the translation. Notice that it is not enough to state that if e evaluates to v then $e^\star$ applied to the same store will give the same value. Indeed, e may not terminate on some stores, while $e^\star$ does. That is why the following theorem is made of two assertions, which really express the semantical equivalence between e and $e^\star$.

Theorem 1 (correctness of the translation) *Let Γ be a typing environment, whose references are $x_1 : \mathsf{ref}\ \tau_1, \ldots, x_n : \mathsf{ref}\ \tau_n$, and let e be an expression such that $\Gamma \vdash e : (\tau, (\rho, \omega))$. Then for any store θ mapping values to the variables x_i, we have*

$$
\forall \theta', v.\quad \theta.e \longmapsto^\star \theta'.v \quad \Longrightarrow \quad (e^\star\ \theta(\rho)) \ \triangleright^\star\ (\theta'(\omega), \textit{val}(\theta',v)^\star)
$$

and

$$
\forall s', v_0.\quad (e^\star\ \theta(\rho)) \ \triangleright^\star\ (s', v_0) \quad \Longrightarrow \quad \exists \theta', v.\ \theta.e \longmapsto^\star \theta'.v
$$

where $\theta(\{x_1,\ldots,x_n\}) \stackrel{def}{=} \{x_1 = \theta(x_1)^\star; \ldots; x_n = \theta(x_n)^\star\}$.

The proof is made by induction over the length of the derivation $\longmapsto^\star$ and by case analysis over e. (For the second assertion, we reason *ad absurdum* and we show that the evaluation of $e^\star$ needs at least as many reduction steps as the evaluation of e.) The complete proofs of the above theorem, and of all the necessary lemmas, are given in [4], pages 32–35, 52–53 and appendix A.

4 Discussion

In this paper, we have combined existing materials, namely monads and effects, and only the combination of both of them as generalized monads is really new. Then we have defined a particular generalized monad and applied it to the translation of imperative programs, which resulted in a very natural translation. For instance, the following program (in which a loop is a special case of recursive function and a sequence a special case of let)

$$\mathsf{while}\ !n > 0\ \mathsf{do}\ s\ := !s + !n;\ n\ := !n - 1\ \mathsf{done}$$

is translated, after some reductions, into the following function

$$\mathsf{rec}\ f\ \{s; n\}\ = \mathsf{if}\ n > 0\ \mathsf{then}\ f\ \{s + n; n - 1\}\ \mathsf{else}\ \{s; n\}$$

where $\{s; n\}$ stands for a two fields record containing the values s and n. The function above is exactly what one would have written, and it is easy to establish logical properties on such interpretations.

Related work. To our knowledge, the only similar work is a paper of Wadler [14], which have been developed independently. In that paper, Wadler combines the effect discipline of Talpin and Jouvelot [12] and the notion of monads, proposing a translation from the first to the second, its proof of correctness and type reconstruction algorithms. The monadic operator appears as parameterized with an effect, but the generality with respect to an abstract notion of effect is not clearly stated, nor the axioms the operators should satisfy. Above all, Wadler proposes a translation whose target is the monadic language, while we propose a translation into purely functional programs using the monadic operations, which are unfolded immediately. Therefore, we never have to type monadic expressions, which would require dependent types. Wadler argues that monads with effects could be used directly to refine the `ST` monad in Haskell [2], but no detail is given about the way to do it, which is not obvious at all without dependent types.

A modular translation. One of the greatest advantage of generalized monads over classical ones is to permit a modular translation. Contrary to the classical notion where the effect is set once for all, we can translate a first program in a given context, then extend that context with additional references and translate a second program that uses the first one without retranslating it. We can also introduce a new kind of effects, as exceptions for instance, and, provided that the new monadic operator is conservative over the first one, we can still reuse previous translations.

Theory and practice. The translation we proposed has been put into practice in the Coq proof assistant [1] to define a method to establish the total correctness of imperative programs [3, 4]. The programming language is a bit more complete than the one presented here, with a simple form of polymorphism, arrays, syntax for loops and sequences, and a large set of predefined constants including arithmetics and boolean connectives. The logical part of that work is beyond the scope of that article but is described in [3, 4]. (It mainly consists in adding some logical informations inside the translation.) It has been applied to the proof of correctness of non-trivial algorithms, such that Knuth-Pratt-Morris or in-place heapsort [4, 5].

Acknowledgments. I wish to thank John C. Reynolds for his help in finding the right set of axioms for generalized monads, and Christine Paulin for many stimulating discussions.

References

[1] The Coq Proof Assistant. `http://coq.inria.fr/`.

[2] Haskell 1.4, a non-strict, purely functional language. Technical report, Yale University, April 1997.

[3] J.-C. Filliâtre. Proof of Imperative Programs in Type Theory. In *International Workshop, TYPES '98, Kloster Irsee, Germany*, volume 1657 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1998.

[4] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types.* PhD thesis, Université Paris-Sud, July 1999.

[5] J.-C. Filliâtre and N. Magaud. Certification of Sorting Algorithms in the System Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.

[6] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Java Series. Sun Microsystems, 1996.

[7] J. Guzmán and A. Suárez. An extended type system for exceptions. In *Record of the fifth ACM SIGPLAN workshop on ML and its Applications*, June 1994. Also appear as Research Report 2265, INRIA, BP 105 - 78153 Le Chesnay Cedex, France.

[8] B. W. Kernighan and R. Pike. *The Practice of Programming.* Addison-Wesley, 1999.

[9] E. Moggi. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, 1989.

[10] E. Moggi. Notions of computations and monads. *Information and Computation*, 93(1), 1991.

[11] J.-P. Talpin and P. Jouvelot. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, 2(3), 1992.

[12] J.-P. Talpin and P. Jouvelot. The Type and Effect Discipline. *Information and Computation*, 111(2):245–296, 1994.

[13] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi*, NATO ASI Series. Springer Verlag, 1993.

[14] P. Wadler. The marriage of effects and monads. In *International Conference on Functional Programming*, pages 63–74, Baltimore, September 1998. ACM.

[15] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115:38–94, 1994.