Patterns for computational effects arising from a monad or a comonad

Jean-Guillaume Dumas, Dominique Duval, Jean-Claude Reynaud

Patterns for computational effects arising from a monad or a comonad

Jean-Guillaume Dumas* Dominique Duval*
Jean-Claude Reynaud†

October 2, 2013

Abstract

We propose patterns for proving properties of programs involving computational effects, in the framework of decorated logics. Although this framework does not mention monads nor comonads, it can be instantiated for a combination of monads and comonads. We propose two dual patterns. The first one provides inference rules which can be interpreted in the Kleisli category of a monad and the coKleisli category of the associated comonad. For instance, in this pattern, the raising of exceptions corresponds to the exception monad (with endofunctor $A + E$) on some category and their handling corresponds to a comonad (with the same endofunctor $A + E$) on the Kleisli category of the monad. In a dual way, the second pattern provides inference rules which can be interpreted in the coKleisli category of a comonad and the Kleisli category of the associated monad. For instance, in this pattern, the lookup operation on states corresponds to the comonad with endofunctor $A \times S$ and the update operation corresponds to a monad on the coKleisli category of the comonad. Each pattern consists in a language with an inference system. This system can be used for proving properties of programs which involve an effect that can be associated to a monad (respectively a comonad). The pattern provides generic rules for dealing with any monad (respectively comonad), and it can be extended with specific rules for each effect.

1 Introduction

Although there is no precise definition of computational effects, it is generally accepted that the mechanism of exceptions in a computer language is an effect,

*Laboratoire J. Kuntzmann, Université de Grenoble. 51, rue des Mathématiques, umr CNRS 5224, bp 53X, F38041 Grenoble, France, {Jean-Guillaume.Dumas,Dominique.Duval}@imag.fr.
†Reynaud Consulting (RC), Jean-Claude.Reynaud@imag.fr.

as well as the evolution of the states of the memory for an imperative language. In order to formalize effects one can choose between types and effects systems [9], monads [10] and their associated Lawvere theories [11], comonads [15], or decorated logics [1]. Each of these approaches rely on some classification of the syntactic expressions according to their interaction with effects. In this paper we use decorated logics which, by extending this classification to equations, they provide a proof system adapted to each effect.

The aim of this paper is to provide two dual patterns for building decorated logics. The first pattern can be used for effects which arise from a monad, and the second for effects which arise from a comonad. Each pattern consists in a set of annotations and a sound inference system which are used to prove properties of programs involving those kinds of effects. Then, for each chosen effect arising from a monad or a comonad, the corresponding pattern can be extended, according to the particular properties of this effect. For instance, we then apply the patterns to two specific effects: the exception effect and the state effect. In particular, we focus on the rules for operators of arbitrary arity and for case distinction, which are expressed categorically as finite products and finite coproducts.

We do not claim that each effect arises either from a monad or from a comonad, but this paper only deals with such effects. Intuitively, an effect which constructs features may arise from a monad, while an effect which observes features may arise from a comonad. However, some interesting features in the monad pattern stem from the well-known fact that each monad determines a comonad on its Kleisli category, and dually for the comonad pattern.

More precisely, on the monads side, let (M, η, μ) be a monad on a category $\mathbf{C}^{(0)}$ and let $\mathbf{C}^{(1)}$ be the Kleisli category of (M, η, μ) on $\mathbf{C}^{(0)}$. Then M can be seen as the endofunctor of a comonad (M, ε, δ) on $\mathbf{C}^{(1)}$, so that we may consider the coKleisli category $\mathbf{C}^{(2)}$ of (M, ε, δ) on $\mathbf{C}^{(1)}$. The canonical functors from $\mathbf{C}^{(0)}$ to $\mathbf{C}^{(1)}$ and from $\mathbf{C}^{(1)}$ to $\mathbf{C}^{(2)}$ give rise to a hierarchy of terms: pure terms in $\mathbf{C}^{(0)}$, constructors in $\mathbf{C}^{(1)}$, modifiers in $\mathbf{C}^{(2)}$. On the comonads side, we get a dual hierarchy: pure terms in $\mathbf{C}^{(0)}$, observers in $\mathbf{C}^{(1)}$, modifiers in $\mathbf{C}^{(2)}$.

We apply this point of view to the study of two fundamental examples of effects: exceptions and states.

Following [10], we consider that the exceptions effect arise from the monad $A+E$ (where E is the set of exceptions), thus a decorated logic for exceptions is built by extending the pattern for monads. The monad itself provides a decoration for the raising operation, which constructs an exception, while the comonad on its Kleisli category provides a decoration for the handling operation.

Following [3], we consider that the states effect arise from the comonad $A \times S$ (where S is the set of states), thus a decorated logic for states is built by extending the pattern for comonads. The comonad itself provides a decoration for the lookup operation, which observes the state, while the monad on its coKleisli category provides a decoration for the update operation.

In fact, the decorated logic for exceptions is not exactly dual to the decorated logic for states, because we assume that the intended interpretation takes place in a distributive category, like the category of sets, which is not codistributive.

Other effects would lead to other additional rules, but we have chosen to focus on two effects which are very well known from various points of view. Our goal is to enligthen the contributions of each approach: the annotation system from the types and effects systems [9], the major role of monads for some effects [10], and the dual role of comonads [15], as well as the flexibility of decorated logics [1].

Section 2 is devoted to the description of our first pattern and of its decorated proof system, valid for any effect arising from a monad. Then, Section 3 applies this pattern to the example of the exception effect. The generic system is completed by adding more specific rules such as the ones dealing with case distinction. Our second generic pattern, fully dual to the first one, is presented in Section 4. It is meant to be applied to any effect arising from a comonad. Finally, in Section 5, we apply the latter pattern to the state effect by adding rules such as the ones dealing with multiple arity functions.

2 Effect based on a monad

Starting with Moggi's seminal paper [10] and its application to Haskell [16], various papers deal with the effects arising from a monad, for instance [11, 13, 8, 12].

2.1 A decorated logic for a monad

In this Section we define a logic L_M, made of a grammar and an inference system. It is called a *decorated* logic because its grammar and inference rules are essentially the grammar and inference rules for a "usual" logic, namely the equational logic with conditionals, together with *decorations* (represented by a superscript) for the terms and the equations. The decorations for terms are similar to the *annotations* of the types and effects systems [9]. In Section 2.2 we associate to each monad a model of this decorated logic, which provides a meaning to the decorations and rules of the logic L_M.

Decorated logics are introduced in [1] in an abstract categorical framework, which will not be explicitly used in this paper.

Grammar of L_M:

Types:	$t ::=$	$A \mid B \mid \cdots \mid t + t \mid \mathbb{0} \mid t \times t \mid \mathbb{1}$
Terms:	$f ::=$	$\downarrow f \mid id_t \mid f \circ f \mid$
		$\langle f, f \rangle \mid pr_{t,t,1} \mid pr_{t,t,2} \mid \langle\,\rangle_t$
		$[f \mid f] \mid in_{t,t,1} \mid in_{t,t,2} \mid [\,]_t \mid$
Equations:	$e ::=$	$f \equiv f$
Decorations:		
for terms:	$(d) ::=$	$(0) \mid (1) \mid (2)$
for equations:	$(d_{eq}) ::=$	$(s) \mid (w)$

Terms are called *pure* when their decoration is (0), they are called *constructors* when their decoration is (1) and they are called *modifiers* when their decoration is (2). Equations are called *strong* when their decoration is (s) and *weak* when their decoration is (w). Since it is important to see clearly the difference between strong and weak equations, we use $\cong$ and $\sim$ instead of $\equiv^{(s)}$ and $\equiv^{(w)}$, respectively.

The inference rules of L_M are given in Figures 1, 2 and 3; the symbol (d), as well as the absence of decoration, stand for "any decoration".

$$\frac{f^{(0)} : A \to B}{f^{(1)} : A \to B} \qquad \frac{f^{(1)} : A \to B}{f^{(2)} : A \to B} \qquad \frac{f^{(2)} : A \to B}{(\downarrow f)^{(1)} : A \to B}$$

$$\frac{A}{id_A^{(0)} : A \to A} \qquad \frac{f^{(d)} : A \to B \quad g^{(d)} : B \to C}{(g \circ f)^{(d)} : A \to C}$$

$$\frac{B_1 \quad B_2}{pr_1^{(0)} : B_1 \times B_2 \to B_1 \quad pr_2^{(0)} : B_1 \times B_2 \to B_2}$$

$$\frac{f_1^{(0)} : A \to B_1 \quad f_2^{(0)} : A \to B_2}{\langle f_1, f_2 \rangle^{(0)} : A \to B_1 \times B_2} \qquad \frac{A}{\langle\,\rangle_A^{(0)} : A \to \mathbb{1}}$$

$$\frac{A_1 \quad A_2}{in_1^{(0)} : A_1 \to A_1 + A_2 \quad in_2^{(0)} : A_2 \to A_1 + A_2}$$

$$\frac{f_1^{(0)} : A_1 \to B \quad f_2^{(0)} : A_2 \to B}{[f_1 | f_2]^{(0)} : A_1 + A_2 \to B} \qquad \frac{B}{[\,]_B^{(0)} : \mathbb{0} \to B}$$

$$\frac{f_1^{(1)} : A_1 \to B \quad f_2^{(1)} : A_2 \to B}{[f_1 | f_2]^{(1)} : A_1 + A_2 \to B}$$

Figure 1: Typing and decorations rules for a monad

$$\frac{f}{\downarrow f \sim f} \qquad \frac{f^{(1)} \sim g^{(1)}}{f \cong g}$$

$$\frac{f\colon A \to B}{f \circ id_A \cong f} \qquad \frac{f\colon A \to B}{id_B \circ f \cong f}$$

$$\frac{f\colon A \to B \;\; g\colon B \to C \;\; h\colon C \to D}{h \circ (g \circ f) \cong (h \circ g) \circ f}$$

$$\frac{f}{f \cong f} \qquad \frac{f \cong g}{g \cong f} \qquad \frac{f \cong g \;\; g \cong h}{f \cong h}$$

$$\frac{f\colon A \to B \;\; g_1 \cong g_2\colon B \to C}{g_1 \circ f \cong g_2 \circ f\colon A \to C}$$

$$\frac{f_1 \cong f_2\colon A \to B \;\; g\colon B \to C}{g \circ f_1 \cong g \circ f_2\colon A \to C}$$

$$\frac{f}{f \sim f} \qquad \frac{f \sim g}{g \sim f} \qquad \frac{f \sim g \;\; g \sim h}{f \sim h}$$

$$\frac{f^{(0)}\colon A \to B \;\; g_1 \sim g_2\colon B \to C}{g_1 \circ f \sim g_2 \circ f}$$

$$\frac{f_1 \sim f_2\colon A \to B \;\; g\colon B \to C}{g \circ f_1 \sim g \circ f_2\colon A \to C}$$

Figure 2: Equational rules for a monad (1)

Remark 2.1.

- The first rules in Figure 1 are conversion rules for terms. The first two conversions are safe: they may be used for upcasting pure terms to constructors and constructors to modifiers. The third conversion is unsafe: each modifier $f^{(2)}$ may be downcasted to a constructor $(\downarrow f)^{(1)}$, but several modifiers may be downcasted to the same constructor. The first rule in Figure 2 says that a modifier is weakly equal to its downcasted constructor, and the next rule is the conversion rule from strong equations to weak ones.

- The decoration rule for composition and the upcasting rules imply the following derived rules, for each decorations $d, d', d'' \in \{0, 1, 2\}$ such that $d'' \geq \max(d, d')$:

$$\frac{f^{(d)}\colon A \to B \quad g^{(d')}\colon B \to C}{(g \circ f)^{(d'')}\colon A \to C}$$

- The pair $\langle f_1, f_2 \rangle$ exists only when f_1 and f_2 are pure, and it is pure.

$$\frac{f_1^{(0)}\colon A \to B_1 \quad f_2^{(0)}\colon A \to B_2}{pr_1 \circ \langle f_1, f_2\rangle \cong f_1 \quad pr_2 \circ \langle f_1, f_2\rangle \cong f_2}$$

$$\frac{g^{(0)}\colon A \to B_1 \times B_2 \quad pr_1 \circ g \cong f_1^{(0)} \quad pr_2 \circ g \cong f_2^{(0)}}{g \cong \langle f_1, f_2\rangle}$$

$$\frac{f^{(0)}\colon A \to \mathbb{1}}{f \cong \langle\,\rangle_A}$$

$$\frac{f_1^{(1)}\colon A_1 \to B \quad f_2^{(1)}\colon A_2 \to B}{[f_1|f_2] \circ in_1 \cong f_1 \quad [f_1|f_2] \circ in_2 \cong f_2}$$

$$\frac{g^{(1)}\colon A_1 + A_2 \to B \quad g \circ in_1 \cong f_1^{(1)} \quad g \circ in_2 \cong f_2^{(1)}}{g \cong [f_1|f_2]}$$

$$\frac{g\colon \mathbb{0} \to B}{g \sim [\,]_B}$$

Figure 3: Equational rules for a monad (2)

- Thanks to the conversion from pure terms to constructors, the copair $[f_1|f_2]$ exists and it is a constructor when f_1 and f_2 are pure terms and constructors, and in addition it is pure when both f_1 and f_2 are pure.
- Strong and weak equality coincide on pure terms and on constructors, they may differ on modifiers.
- Weak equations form a "weak" congruence, in the sense that the substitution rule for weak equations holds only when the substituted term is pure.
- With these decorated rules it is easy to prove that:

$$f \sim g \text{ if and only if } \downarrow f \cong \downarrow g$$

2.2 A decorated model for a monad

Let (M, η, μ) be a monad on a category $\mathbf{C}$ which satisfies the mono requirement, which means that $\eta_A : A \to MA$ is a monomorphism for each object A. In addition, let us assume that $\mathbf{C}$ has finite products and finite coproducts. Then we get a model $\mathbf{C}_M$ of the decorated logic L_M in the following way.

- the types are interpreted as the objects of $\mathbf{C}$;
- the operations and terms are interpreted as morphisms of $\mathbf{C}$, in the following way:
  - a pure term $f^{(0)}\colon A \to B$ as a morphism $f\colon A \to B$ in $\mathbf{C}$;

  - a constructor $f^{(1)} : A \to B$ as a morphism $f : A \to MB$ in $\mathbf{C}$;
  - a modifier $f^{(2)} : A \to B$ as a morphism $f : MA \to MB$ in $\mathbf{C}$;

- the equations between modifiers are interpreted as equalities in $\mathbf{C}$, in the following way:
  - a strong equation $f^{(2)} \cong g^{(2)} : A \to B$ as an equality $f = g : MA \to MB$ in $\mathbf{C}$;
  - a weak equation $f^{(2)} \sim g^{(2)} : A \to B$ as an equality $f \circ \eta_A = g \circ \eta_A : A \to MB$ in $\mathbf{C}$;

- the conversions are interpreted as follows:
  - from pure terms to constructors: $f : A \to B$ is upcasted as $\eta_B \circ f : A \to MB$;
  - from constructors to modifiers: $f : A \to MB$ is upcasted as $\mu_B \circ Mf : MA \to MB$;
  - from modifiers to constructors: $f : MA \to MB$ is downcasted as $\downarrow f = f \circ \eta_A : A \to MB$;

- for identities and composition:
  - the identity $id^{(0)}_A : A \to A$ is interpreted as $id_A : A \to A$ in $\mathbf{C}$;
  - the composition of two modifiers $f^{(2)} : A \to B$ and $g^{(2)} : B \to C$ is interpreted as $g \circ f : MA \to MB$ in $\mathbf{C}$;

- products are interpreted as follows:
  - the unit type as the final object of $\mathbf{C}$;
  - the term $\langle\,\rangle_A : A \to \mathbb{1}$ as the unique morphism from A to $\mathbb{1}$ in $\mathbf{C}$;
  - the product $B_1 \times B_2$ with its projections $pr^{(0)}_1$ and $pr^{(0)}_2$ as the binary product in $\mathbf{C}$;
  - the pair of $f^{(0)}_1 : A \to B_1$ and $f^{(0)}_2 : A \to B_2$ as the pair $\langle f_1, f_2 \rangle : A \to B_1 \times B_2$ in $\mathbf{C}$;

- coproducts are interpreted as follows:
  - the empty type as the initial object of $\mathbf{C}$;
  - the term $[\,]_A : \mathbb{0} \to A$ as the unique morphism from $\mathbb{0}$ to A in $\mathbf{C}$;
  - the coproduct $A_1 + A_2$ with its coprojections $in^{(0)}_1$ and $in^{(0)}_2$ as the binary coproduct in $\mathbf{C}$.
  - the copair of $f^{(0)}_1 : A_1 \to B$ and $f^{(0)}_2 : A_2 \to B$ as the copair $[f_1|f_2] : A_1 + A_2 \to B$ in $\mathbf{C}$;
  - the copair of $f^{(1)}_1 : A_1 \to B$ and $f^{(1)}_2 : A_2 \to B$ as the copair $[f_1|f_2] : A_1 + A_2 \to MB$ in $\mathbf{C}$;

The interpretation of composition and equations is defined above only for modifiers. The general definition, for terms with arbitrary decorations, is the following one: first all terms are converted to modifiers, then the definition above is used. It is easy to check that there are "obvious" shortcuts: the composition of two constructors $f^{(1)}\colon A \to B$ and $g^{(1)}\colon B \to C$ is the Kleisli composition of $f\colon A \to MB$ and $g\colon B \to MC$, the composition of two pure terms $f^{(0)}\colon A \to B$ and $g^{(0)}\colon B \to C$ is the composition of $f\colon A \to B$ and $g\colon B \to C$ in $\mathbf{C}$, an equation $f^{(1)} \cong g^{(1)}\colon A \to B$ or $f^{(1)} \sim g^{(1)}\colon A \to B$ is $f = g\colon A \to MB$ in $\mathbf{C}$ and an equation $f^{(0)} \cong g^{(0)}\colon A \to B$ or $f^{(0)} \sim g^{(0)}\colon A \to B$ is $f = g\colon A \to B$ in $\mathbf{C}$.

Now, it is easy to check that $\mathbf{C}_M$ is indeed a model of L_M.

Proposition 2.2. *The rules of the logic L_M are satisfied by $\mathbf{C}_M$.*

3 Exceptions

3.1 Operations on exceptions

In Section 2.2 we have defined a model $\mathbf{C}_M$ of the logic L_M from Section 2.1, for any monad (M, η, μ) satisfying the mono requirement on any category $\mathbf{C}$ with finite products and coproducts.

For a more specific choice of the category and the monad, it may happen that additional rules can be added to the logic L_M. In this Section we consider a category $\mathbf{C}$ with finite products and coproducts such that the coprojections are monomorphisms and with a distinguished object E called the *object of exceptions*. The *monad of exceptions* on $\mathbf{C}$ is the monad (M, η, μ) with endofunctor $MA = A + E$, its unit η is made of the coprojections $\eta_A\colon A \to A + E$, and its multiplication μ "merges" the exceptions, in the sense that $\mu_A = [id_{A+E}|in_A]\colon (A + E) + E \to A + E$ where $in_A\colon E \to A + E$ is the coprojection.

We define a logic L_{exc} by extending L_M. For each set *Exn* of *exception names* and each family of objects $(V_T)_{T \in Exn}$ in $\mathbf{C}$ we build a model $\mathbf{C}_{exc}$ of L_{exc}. In this context, the *exceptions of name* T are obtained by *tagging*, or encapsulating, the values in V_T. The model $\mathbf{C}_{exc}$ extends the model $\mathbf{C}_M$ of L_M with functions for raising and handling the exceptions of name T for each $T \in Exn$.

If can be assumed that there is a hierarchy of exception names, then the rules must be be slightly modified.

Since coproducts form an associative and commutative law, we have new rules for coproducts when the monad is $MA = A+E$: we have a *left copair* $[f_1|f_2]_l^{(2)}$ of a constructor $f_1^{(1)}$ and a modifier $f_2^{(2)}$, satisfying the first three rules in Figure 4. There are also three rules (omitted), symmetric to these ones, for the *right copair* $[f_1|f_2]_r^{(2)}$ of a modifier $f_1^{(2)}$ and a constructor $f_2^{(1)}$. The last rule in Figure 4

expresses the fact that, when $MA = A + E$, two modifiers coincide as soon as they coincide on ordinary values and on exceptions.

$$\frac{f_1^{(1)} \colon A_1 \to B \quad f_2^{(2)} \colon A_2 \to B}{[f_1|f_2]_l^{(2)} \colon A_1 + A_2 \to B}$$

$$\frac{f_1 \colon A_1 \to B \;\; f_2 \colon A_2 \to B}{[f_1|f_2]_l \circ in_1 \sim f_1 \quad [f_1|f_2]_l \circ in_2 \cong f_2}$$

$$\frac{g^{(2)} \colon A_1 + A_2 \to B \;\; g \circ in_1 \sim f_1^{(1)} \;\; g \circ in_2 \cong f_2^{(2)}}{g \cong [f_1|f_2]_l}$$

$$\frac{f, g \colon A \to B \quad f \sim g \quad f \circ [\,]_A \cong g \circ [\,]_A}{f \cong g}$$

Figure 4: Exceptions: additional rules for coproducts

Remark 3.1. For instance, the coproduct of any type A with the empty type $\mathbb{0}$ is isomorphic to A, with coprojections $id_A^{(0)} : A \to A$ and $[\,]_A^{(0)} : \mathbb{0} \to A$. It gives rise to the left copair $[f_1|f_2]_l^{(2)} : A \to B$ of any constructor $f_1^{(1)} \colon A \to B$ with any modifier $f_2^{(2)} \colon \mathbb{0} \to B$. This property will be used in the construction of the `try/catch` expressions.

We distinguish two languages for exceptions, as in [3]:

- The *core* language performs the core operations of *tagging*, which encapsulates an ordinary value into an exception, and *untagging*, which recovers the ordinary value encapsulated in an exception. This language is *private*.
- The *programmer's* language is *public*. It provides the operations for *raising* and *handling* exceptions, which are defined in terms of the core operations.

The core language. The decorations are the same as for L_M, but their name is adapted to the monad of exceptions:

- A constructor $f^{(1)}$ is called a *propagator*: it may raise an exception but cannot recover from an exception, so that it has to propagate all exceptions.
- A modifier $f^{(2)}$ is called a *catcher*.

Grammar for the core language (in addition to the grammar of L_M):

Types:	$t ::=$	V_T	for each $T \in Exn$
Terms:	$f ::=$	$\mathtt{tag}_T \mid \mathtt{untag}_T$	for each $T \in Exn$

Specific rules for the core language (in addition to the rules of L_M): see Figure 5.

Remark 3.2.

- The weak equations relating $\mathtt{untag}_T$ and $\mathtt{tag}_R$ (for $R = T$ and for $R \neq T$) mean that $\mathtt{untag}_T$, when applied to an exception, recovers the argument of the exception if this exception has name T and propagates the exception otherwise.
- The last rule in Figure 5 is the *local-to-global* rule which asserts that two functions without argument coincide as soon as they coincide on each exception. It follows that two functions with an argument coincide as soon as they coincide on their argument and on each exception: this is the following rule, which is derived from the local-to-global rule and the last rule in Figure 4.

$$\frac{f, g \colon A \to B \quad f \sim g \qquad \text{for all } T \quad f \circ [\,]_A \circ \mathtt{tag}_T \sim g \circ [\,]_A \circ \mathtt{tag}_T}{f \cong g}$$

$$\frac{T \in \mathit{Exn}}{\mathtt{tag}_T^{(1)} \colon V_T \to \mathbb{0}} \qquad \frac{T \in \mathit{Exn}}{\mathtt{untag}_T^{(2)} \colon \mathbb{0} \to V_T}$$

$$\frac{T \in \mathit{Exn}}{\mathtt{tag}_T \circ \mathtt{untag}_T \sim \mathit{id}_{V_T}} \qquad \frac{T \neq R \in \mathit{Exn}}{\mathtt{tag}_R \circ \mathtt{untag}_T \sim \mathtt{tag}_R \circ [\,]_{V_T}}$$

$$\frac{f, g \colon \mathbb{0} \to B \quad \text{for all } T \in \mathit{Exn}\ f \circ \mathtt{tag}_T \sim g \circ \mathtt{tag}_T}{f \cong g}$$

Figure 5: Exceptions: specific rules for the core language

The programmer's language. The programmer's language for exceptions has no catcher: the only way to catch an exception is by using a `try/catch` expression, which itself propagates exceptions. Thus, the programmer's language has less decorations than the core language: its operations are either pure or constructors. Since weak and strong equations coincide on constructors we may consider that all the equations for the programmer's language are strong. Let $L_M^{(1)}$ be the part of the decorated logic for a monad which deals only with pure terms, constructors and strong equations.

The grammar for the programmer's language does not include the private tagging and untagging operations, but the public `throw` and `try/catch` constructions.

Grammar for the programmer's language (in addition to the grammar of $L_M^{(1)}$):

$$\begin{array}{lll} \text{Types:} & t ::= & V_T \quad \text{for each } T \in \mathit{Exn} \\ \text{Terms:} & f ::= & \mathtt{throw}_{t,T} \mid \\ & & \mathtt{try}(f)\mathtt{catch}(T_1 \Rightarrow f \mid T_n \Rightarrow f) \\ & & \quad \text{for each } T, T_1, \ldots, T_n \in \mathit{Exn} \text{ (with } n > 0) \end{array}$$

The rules for the programmer's language (in addition to the rules of $L_{M,1}$) are only the typing and decorating rules for the `throw` and `try`/`catch` expressions, see Figure 6.

$$\frac{T \in \mathit{Exn} \quad B}{\mathtt{throw}_{B,T}^{(1)} : V_T \to B}$$

$$\frac{f^{(1)} : A \to B \quad g_1^{(1)} : V_{T_1} \to B \quad \ldots \quad g_n^{(1)} : V_{T_n} \to B}{(\mathtt{try}(f)\mathtt{catch}(T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n))^{(1)} : A \to B}$$

Figure 6: Exceptions: rules for the programmer's language

Definitions of `throw` and `try`/`catch` The `throw` and `try`/`catch` expressions in the programmer's language are defined in terms of the `tag` and `untag` core operations: see Definition 3.3 for `throw`, Definition 3.4 for `try`/`catch` when only one exception is catched, and more generally Definition 3.6 for the general `try`/`catch` construction (so that Definition 3.4 is redundant). Diagrams are provided for illustrating the Definitions; some subscripts and decorations are omitted.

Definition 3.3. For each type B and each $T \in \mathit{Exn}$, the propagator $\mathtt{throw}_{B,T}^{(1)} : V_T \to B$ is:

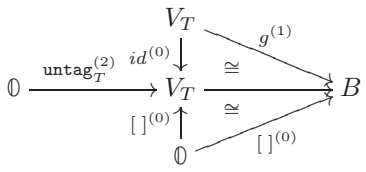
$$\mathtt{throw}_{B,i}^{(1)} = [\,]_B^{(0)} \circ \mathtt{tag}_T^{(1)}$$

$$V_T \xrightarrow{\mathtt{tag}_T^{(1)}} \mathbb{0} \xrightarrow{[\,]_B^{(0)}} B$$

This means that raising an exception with name T consists in tagging the given ordinary value (in V_T) as an exception and coerce it to any given type B.

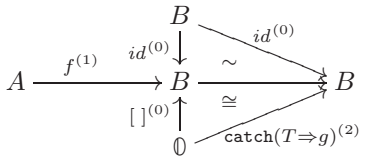
Definition 3.4. For each type B, each $T \in \mathit{Exn}$ and each propagator $g^{(1)} : V_T \to B$, the catcher $(\mathtt{catch}(T \Rightarrow g))^{(2)} : \mathbb{0} \to B$ is:

$$\mathtt{catch}(T \Rightarrow g) = [\, g^{(1)} \mid [\,]_B^{(0)} \,]^{(1)} \circ \mathtt{untag}_T^{(2)}$$



then for each propagator $f^{(1)} \colon A \to B$, the catcher $(\mathtt{TRY}(f)\mathtt{catch}(T \Rightarrow g))^{(2)} \colon A \to B$ is:

$$\mathtt{TRY}(f)\mathtt{catch}(T \Rightarrow g) = [\ id_B \mid \mathtt{catch}(T \Rightarrow g)\ ]_l^{(2)} \circ f^{(1)}$$



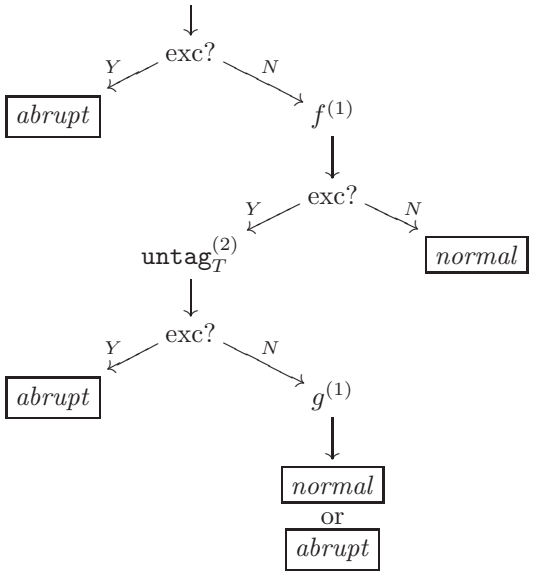
and finally the propagator $(\mathtt{try}(f)\mathtt{catch}(T \Rightarrow g))^{(1)} \colon A \to B$ is:

$$\mathtt{try}(f)\mathtt{catch}(T \Rightarrow g) = \downarrow(\mathtt{TRY}(f)\mathtt{catch}(T \Rightarrow g))$$

Remark 3.5.

- The intermediate expressions $(\mathtt{catch}(T \Rightarrow g))$ and $\mathtt{TRY}(f)\mathtt{catch}(T \Rightarrow g)$ are catchers, they are not part of the programmer's language.
- The expression $\mathtt{try}(f)\mathtt{catch}(T \Rightarrow g)$ is a propagator: the downcast operator prevents it from catching exceptions with name T which might have been raised before the $\mathtt{try}(f)\mathtt{catch}(T \Rightarrow g)$ expression is considered.

Definition 3.4 corresponds to the following control flow, where `exc?` means "*is this value an exception?*", an *abrupt* termination returns an uncaught exception and a *normal* termination returns an ordinary value. This corresponds, for instance, to the Java mechanims for exceptions [6, 7].



In the general case, we use the abbreviated notation:

$$\mathtt{catch}(T_i \Rightarrow g_i)_{p\le i\le n} = \mathtt{catch}(T_p \Rightarrow g_p | \dots | T_n \Rightarrow g_n)$$

Definition 3.6. For each type B, each list $(T_1, \dots, T_n)$ (with $n \ge 1$) of exceptional names and each propagators

$$g_1^{(1)} \colon V_{T_1} \to B, \dots, g_n^{(1)} \colon V_{T_n} \to B$$

the family of catchers

$$k_p^{(2)} = (\mathtt{catch}(T_i \Rightarrow g_i)_{p\le i\le n})^{(2)} \colon \mathbb{0} \to B$$

(for $p = 1, \dots, n$) is defined recursively by:

$$k_p^{(2)} \;=\; \begin{cases} [\ g_p^{(1)} \mid [\ ]_B^{(0)}\ ]^{(1)} \circ \mathtt{untag}_{T_p}^{(2)} & \text{when } p = n \\ [\ g_p^{(1)} \mid k_{p+1}^{(2)}\ ]_l^{(2)} \circ \mathtt{untag}_{T_p}^{(2)} & \text{when } p < n \end{cases}$$

then for each propagator $f^{(1)} \colon A \to B$, the catcher
$(\mathtt{TRY}(f)\mathtt{catch}(T_i \Rightarrow g_i)_{1\le i\le n})^{(2)} \colon A \to B$ is:

$$\mathtt{TRY}(f)\mathtt{catch}(T_i \Rightarrow g_i)_{1\le i\le n} = [\ id_B \mid k_1\ ]_l^{(2)} \circ f^{(1)}$$

and finally the propagator
$(\mathtt{try}(f)\mathtt{catch}(T_i \Rightarrow g_i)_{1\le i\le n})^{(1)} \colon A \to B$ is:

$$\mathtt{try}(f)\mathtt{catch}(T_i \Rightarrow g_i)_{1\le i\le n} =$$
$$\downarrow(\mathtt{TRY}(f)\mathtt{catch}(T_i \Rightarrow g_i)_{1\le i\le n})$$

3.2 Case distinction and sequential product, for exceptions

Since the programmer's language for exceptions has no catcher, the coproducts of constructors, which are valid for any monad, provide case distinction for all terms in this language.

There is no binary product of propagators. Indeed, if $f_1^{(1)}\colon A \to B_1$ and $f_2^{(1)}\colon A \to B_2$ both raise an exception, it is in general impossible to find $f^{(1)}\colon A \to B_1 \times B_2$ such that $pr_1 \circ f \cong f_1$ and $pr_2 \circ f \cong f_2$. However, there are several ways to formalize the fact of first evaluating f_1 then f_2: for instance by using a strong monad [10], or a sequential product [4], or productors [14].

We define the *left sequential product* of f_1 and f_2 from *semi-pure products*, as in [4]; the *right sequential product* can be defined in a symmetric way.

For this purpose, let us introduce a third decorations for equations: (p) and let us focus on the interpretation in the category of sets. The interpretation of $\equiv^{(p)}$ is defined from the usual ordering between partial functions: each $f\colon A \to B+E$ determines a partial function $\tilde{f}\colon A \rightharpoonup B$ with domain of definition $\mathbf{D}(\tilde{f}) = \{x \in A \mid f(x) \in B\}$, such that $\tilde{f}$ coincides with f on $\mathbf{D}(\tilde{f})$. Then for $f, g\colon A \to B+E$ we say that $f \equiv^{(p)} g$ when $\tilde{f} \leq \tilde{g}$ in the sense of partial functions. Thus, the relation $\equiv^{(p)}$ is a preorder compatible with composition, but it is not symmetric. For readability, we use $\preccurlyeq$ instead of $\equiv^{(p)}$.

$$\frac{f}{f \preccurlyeq f} \qquad \frac{f \preccurlyeq g \quad g \preccurlyeq h}{f \preccurlyeq h}$$

$$\frac{f\colon A \to B \quad g_1 \preccurlyeq g_2\colon B \to C}{g_1 \circ f \preccurlyeq g_2 \circ f\colon A \to C}$$

$$\frac{f_1 \preccurlyeq f_2\colon A \to B \quad g\colon B \to C}{g \circ f_1 \preccurlyeq g \circ f_2\colon A \to C}$$

$$\frac{f_1^{(0)}\colon A \to B_1 \quad f_2^{(1)}\colon A \to B_2}{\langle f_1, f_2\rangle_l^{(1)}\colon A \to B_1 \times B_2}$$

$$\frac{f_1\colon A \to B_1 \quad f_2\colon A \to B_2}{pr_1 \circ \langle f_1, f_2\rangle_l \preccurlyeq f_1 \quad pr_2 \circ \langle f_1, f_2\rangle_l \cong f_2}$$

$$\frac{g^{(1)}\colon A \to B_1 \times B_2 \quad pr_1 \circ g \preccurlyeq f_1^{(0)} \quad pr_2 \circ g \cong f_2^{(1)}}{g \cong \langle f_1, f_2\rangle_l}$$

Figure 7: Exceptions: additional rules for products

The left product $\langle f_1, f_2\rangle_l^{(1)}$ of a pure term $f_1^{(0)}$ and a propagator $f_2^{(1)}$ is a propagator, characterized by the last rules in Figure 7. The right product $\langle f_1, f_2\rangle_r^{(1)}$ of a propagator $f_1^{(1)}$ and a pure term $f_2^{(0)}$ is characterized by the dual rules.

Now we can define the *left sequential product* of two propagators $f_1^{(1)} \colon A \to B_1$ and $f_2^{(1)} \colon A \to B_2$, corresponding to first evaluating f_1 then f_2, as the following propagator $\langle f_1, f_2\rangle_l^{(1)} \colon A \to B_1 \times B_2$ (where $q_1 \colon B_1 \times A \to B_1$ is the projection):

$$\langle f_1^{(1)}, f_2^{(1)}\rangle_l^{(1)} = \langle q_1^{(0)}, f_2^{(1)}\rangle_l \circ \langle f_1^{(1)}, id_A^{(0)}\rangle_r$$

4 Effect based on a comonad

Effects arising from a comonad are studied for instance by [15, 5].

4.1 A decorated logic for a comonad

The dual of the logic L_M for monads, from Section 2.1, is a logic L_T for comonads. It follows that:

- The grammar of L_T is the same as the grammar of L_M,
  but a term with decoration (1) is now called an *observer*.
- The typing and decorations rules of L_T are nearly the same as the corresponding rules for L_M, in Figure 1, except for the last rule:

  $$\frac{f_1^{(1)} \colon A_1 \to B \quad f_2^{(1)} \colon A_2 \to B}{[f_1|f_2]^{(1)} \colon A_1 + A_2 \to B}$$

  which is replaced by:

  $$\boxed{\frac{f_1^{(1)} \colon A \to B_1 \quad f_2^{(1)} \colon A \to B_2}{\langle f_1, f_2\rangle^{(1)} \colon A \to B_1 \times B_2}}$$

- The equational rules for L_M in Figure 2 are valid for L_T, except for the last two rules which are replaced by:

  $$\boxed{\begin{array}{c}\dfrac{f \colon A \to B \quad g_1 \sim g_2 \colon B \to C}{g_1 \circ f \sim g_2 \circ f} \\[2ex] \dfrac{f_1 \sim f_2 \colon A \to B \quad g^{(0)} \colon B \to C}{g \circ f_1 \sim g \circ f_2 \colon A \to C}\end{array}}$$

  This means that weak equations form a "weak" congruence, in the sense that the replacement rule for weak equations holds only when the replaced term is pure.

- The equational rules for L_M in Figure 3 are replaced by the dual rules for L_T in Figure 8. This means that the decorations for binary products switch from (0) to (1) and the decorations for binary coproducts from (1) to (0).

$$\frac{f_1^{(1)} \colon A \to B_1 \quad f_2^{(1)} \colon A \to B_2}{pr_1 \circ \langle f_1, f_2 \rangle \cong f_1 \quad pr_2 \circ \langle f_1, f_2 \rangle \cong f_2}$$

$$\frac{g^{(1)} \colon A \to B_1 \times B_2 \quad pr_1 \circ g \cong f_1^{(1)} \quad pr_2 \circ g \cong f_2^{(1)}}{g \cong \langle f_1, f_2 \rangle}$$

$$\frac{f^{(0)} \colon A \to \mathbb{1}}{f \cong \langle\,\rangle_A}$$

$$\frac{f_1^{(0)} \colon A_1 \to B \quad f_2^{(0)} \colon A_2 \to B}{[f_1|f_2] \circ in_1 \cong f_1 \quad [f_1|f_2] \circ in_2 \cong f_2}$$

$$\frac{g^{(0)} \colon A_1 + A_2 \to B \quad g \circ in_1 \cong f_1^{(0)} \quad g \circ in_2 \cong f_2^{(0)}}{g \cong [f_1|f_2]}$$

$$\frac{g \colon \mathbb{0} \to B}{g \sim [\,]_B}$$

Figure 8: Equational rules for a comonad (2)

4.2 A decorated model for a comonad

This Section is dual to Section 2.2. Let $\mathbf{C}$ be a category with finite products and finite coproducts and (T, ε, δ) a comonad on $\mathbf{C}$ satisfying the epi requirement, which means that $\varepsilon_A : TA \to A$ is an epimorphism for each object A. Then we get a model $\mathbf{C}_T$ of the decorated logic L_T in a way which is similar to Section 2.2, except for the following points.

- terms:
  - an observer constructor $f^{(1)} \colon A \to B$ is interpreted as a morphism $f \colon TA \to B$ in $\mathbf{C}$;
- equations:
  - a weak equation $f^{(2)} \sim g^{(2)} \colon A \to B$ is interpreted as an equality $\varepsilon_B \circ f = \varepsilon_B \circ g \colon TA \to B$ in $\mathbf{C}$;
- conversions:
  - $f \colon A \to B$ is upcasted as $f \circ \varepsilon_A \colon TA \to B$;

- $f\colon TA \to B$ is upcasted as $Tf \circ \delta_A\colon TA \to TB$;
- $f\colon TA \to TB$ is downcasted as $\downarrow f = \varepsilon_B \circ f\colon TA \to B$;

- products:
  - the pair of $f_1^{(1)}\colon A \to B_1$ and $f_2^{(1)}\colon A \to B_2$ is interpreted as the pair $\langle f_1, f_2\rangle\colon TA \to B_1 \times B_2$ in $\mathbf{C}$;
- coproducts:
  - there is no copair of $f_1^{(1)}\colon A_1 \to B$ and $f_2^{(1)}\colon A_2 \to B$.

Clearly, the result dual to Proposition 2.2 holds.

Proposition 4.1. *The rules of the logic L_T are satisfied by $\mathbf{C}_T$.*

5 States

5.1 Operations on states

In this Section we consider a category $\mathbf{C}$ with finite products and coproducts such that the projections are epimorphisms and with a distinguished object S called the *object of states*. We consider the comonad (T, ε, δ) with endofunctor $TA = A \times S$, with counit ε made of the projections $\varepsilon_A\colon A \times S \to A$, and with comultiplication δ which "duplicates" the states, in the sense that $\delta_A = \langle id_{A\times S} | pr_A\rangle\colon A \times S \to (A \times S) \times S$ where $pr_A\colon A \times S \to A$ is the projection.

We call this comonad the *comonad of states*. It is sometimes called the *product comonad*, and it is different from the *costates comonad* or *stores comonad* with endofuntor $TA = S \times A^S$ [5].

We define a logic L_{st} by extending L_T. For each set *Loc* of *locations* (or identifiers) and each family of objects $(V_X)_{X\in Loc}$ in $\mathbf{C}$ we build a model $\mathbf{C}_{st}$ of L_{st}. The model $\mathbf{C}_{st}$ extends the model $\mathbf{C}_T$ of L_T with functions for looking up and updating the locations $X \in Loc$.

The logic we get, and its model, are essentially the same as in [2]. However our approach, via comonads, is new and can be adapted to other comonads.

There is no need here to distinguish a core language from a programmer's language; the unique language for states is dual to the core language for exceptions.

Grammar for the language (in addition to the grammar of L_T):

$$\begin{array}{lll} \text{Types:} & t ::= & V_X \quad \text{for each } X \in Loc \\ \text{Terms:} & f ::= & \mathtt{lookup}_X \mid \mathtt{update}_X \quad \text{for each } X \in Loc \end{array}$$

Since products form an associative and commutative law, we have new rules for products when the comonad is $TA = A \times S$: we have a *left pair* $\langle f_1, f_2 \rangle_l^{(2)}$ of an observer $f_1^{(1)}$ and a modifier $f_2^{(2)}$, satisfying the first three rules in Figure 9. There are also symmetric rules for the *right pair* $\langle f_1 | f_2 \rangle_r^{(2)}$ of a modifier $f_1^{(2)}$ and an observer $f_2^{(1)}$. The last rule in Figure 9 expresses the fact that two modifiers coincide as soon as they return the same result and modify the state in the same way.

$$\frac{f_1^{(1)} \colon A \to B_1 \quad f_2^{(2)} \colon A \to B_2}{\langle f_1, f_2 \rangle_l^{(2)} \colon A \to B_1 \times B_2}$$

$$\frac{f_1^{(1)} \colon A \to B_1 \quad f_2^{(2)} \colon A \to B_2}{pr_1 \circ \langle f_1, f_2 \rangle_l \sim f_1 \quad pr_2 \circ \langle f_1, f_2 \rangle_l \cong f_2}$$

$$\frac{g^{(2)} \colon A \to B_1 \times B_2 \quad pr_1 \circ g \sim f_1^{(1)} \quad pr_2 \circ g \cong f_2^{(2)}}{g \cong \langle f_1, f_2 \rangle_l}$$

$$\frac{f, g \colon A \to B \quad f \sim g \quad \langle\,\rangle_A \circ f \cong \langle\,\rangle_A \circ g}{f \cong g}$$

Figure 9: States: additional rules for products

Specific rules for the language of states are given in Figure 10.

$$\frac{X \in Loc}{\mathtt{lookup}_X^{(1)} \colon \mathbb{1} \to V_X} \qquad \frac{X \in Loc}{\mathtt{update}_X^{(2)} \colon V_X \to \mathbb{1}}$$

$$\frac{X \in Loc}{\mathtt{lookup}_X \circ \mathtt{update}_X \sim id_{V_X}}$$

$$\frac{X \neq Y \in Loc}{\mathtt{lookup}_j \circ \mathtt{update}_X \sim \mathtt{lookup}_Y \circ \langle\,\rangle_{V_X}}$$

$$\frac{f, g \colon \mathbb{0} \to B \quad \text{for all } X \in Loc\ \mathtt{lookup}_X \circ f \sim \mathtt{lookup}_X \circ g}{f \cong g}$$

Figure 10: States: specific rules for the language

The last rule in Figure 10 is the *local-to-global* rule which asserts that two functions without result coincide as soon as they coincide on each location. Together with the last rule in Figure 9, this rule implies that two functions

coincide as soon as they return the same value and coincide on each location.

$$\frac{f, g\colon A \to B \quad f \sim g \qquad \text{for all } X \quad \mathtt{lookup}_X \circ \langle\,\rangle_B \circ f \sim \mathtt{lookup}_X \circ \langle\,\rangle_B \circ g}{f \cong g}$$

Now, in addition, let us assume that the category $\mathbf{C}$ is *distributive*. Then there are additional rules for coproducts, given in Figure 11. These rules are not dual to rules for exceptions, because we have not assumed in Section 3 that the category $\mathbf{C}$ was codistributive.

The interpretation of the modifier $[f_1|f_2]$, when both f_1 and f_2 are modifiers, is the composition of $[f_1|f_2]\colon (A_1 \times S) + (A_2 \times S) \to B \times S$ with the inverse of the canonical morphism $(A_1 \times S) + (A_2 \times S) \to (A_1 + A_2) \times S$: this inverse exists because $\mathbf{C}$ is distributive.

$$\frac{f_1^{(2)}\colon A_1 \to B \quad f_2^{(2)}\colon A_2 \to B}{[f_1|f_2]^{(2)}\colon A_1 + A_2 \to B}$$

$$\frac{f_1\colon A_1 \to B \quad f_2\colon A_2 \to B}{[f_1|f_2] \circ in_1 \cong f_1 \quad [f_1|f_2] \circ in_2 \cong f_2}$$

$$\frac{g^{(2)}\colon A_1 + A_2 \to B \quad g \circ in_1 \cong f_1^{(2)} \quad g \circ in_2 \cong f_2^{(2)}}{g \cong [f_1|f_2]}$$

Figure 11: States: additional rules for coproducts

5.2 Case distinction and sequential product, for states

As soon as the category $\mathbf{C}$ is distributive, the additional rules for coproducts in Figure 11 provide case distinction for the language for states.

There is no binary product of modifiers, but there is a left product of a constructor and a modifier, see Figure 9. Dually there is a right product of a modifier and a constructor.

It follows that the *left and right sequential products* can be defined from these products in a similar way as for exceptions, in Section 3.2.

A major distinction between exceptions and states is that, for exceptions, we have introduced a third decoration for equality, whereas this is not necessary for states. This is due to the introduction of the intermediate notion of *observers* between pure terms (or *values*) and modifiers (or *computations*). This is also a distinction, for states, between the usual approach with the strong monad

of states $M(A) = (A \times S)^S$ and this approach with the comonad of states $T(A) = A \times S$.

6 Conclusion

We have presented two patterns giving sound inference systems to effects arising from a monad or a comonad. We have shown that the obtained inference systems are sound and that they provide a way to handle proofs of programs dealing with such effects.

We also gave detailed examples of applications of these patterns to the state and the exception effects. The obtained decorated proof system for states, as described in Section 4 and similar to the one of [2], has been implemented in Coq [1] so that the given proofs can be automatically verified. ¿From this implementation, it should be possible to extract the generic part corresponding to the comonad pattern, dualize it and extend it to handle e.g. the system for exceptions of Section 2.

A major issue is then the combination of effects. It is indeed required of any approach for effects to be scalable. Within the framework of this paper, it may seem difficult to guess how several effects arising from either monads or comonads can be combined. However, as mentioned in the Introduction, this paper deals with two patterns for instanciating the more general framework of decorated logics [1]. Decorated logics are based on spans in a relevant category of logics, so that the combination of effects can be based on the well-known composition of spans.

References

[1] César Domínguez, Dominique Duval. Diagrammatic logic applied to a parameterization process. Mathematical Structures in Computer Science 20, p. 639-654 (2010).

[2] Jean-Guillaume Dumas, Dominique Duval, Laurent Fousse, Jean-Claude Reynaud. Decorated proofs for computational effects: States. ACCAT 2012. Electronic Proceedings in Theoretical Computer Science 93, p. 45-59 (2012).

[3] Jean-Guillaume Dumas, Dominique Duval, Laurent Fousse, Jean-Claude Reynaud. A duality between exceptions and states. Mathematical Structures in Computer Science 22, p. 719-722 (2012).

[1] Effect categories and COQ, http://coqeffects.forge.imag.fr

[4] Jean-Guillaume Dumas, Dominique Duval, Jean-Claude Reynaud. Cartesian effect categories are Freyd-categories. Journal of Symbolic Computation 46, p. 272-293 (2011).

[5] Jeremy Gibbons, Michael Johnson. Relating Algebraic and Coalgebraic Descriptions of Lenses BX 2012. ECEASST 49 (2012).

[6] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. The Java Language Specification, Third Edition. Addison-Wesley Longman (2005).

[7] Bart Jacobs. A Formalisation of Java's Exception Mechanism. ESOP 2001. Springer Lecture Notes in Computer Science 2028. p. 284-301 (2001).

[8] Paul Blain Levy. Monads and adjunctions for global exceptions. MFPS 2006. Electronic Notes in Theoretical Computer Science 158, p. 261-287 (2006).

[9] John M. Lucassen, David K. Gifford. Polymorphic Effect Systems POPL 1988. ACM Press, p. 47-57 (1988). BibTeX — BibTeX (beta) — EndNote — ACM Ref

[10] Eugenio Moggi. Notions of Computation and Monads. Information and Computation 93(1), p. 55-92 (1991).

[11] Gordon D. Plotkin, John Power. Notions of Computation Determine Monads. FoSSaCS 2002. Springer-Verlag Lecture Notes in Computer Science 2303, p. 342-356 (2002).

[12] Gordon D. Plotkin, Matija Pretnar. Handlers of Algebraic Effects. ESOP 2009. Springer-Verlag Lecture Notes in Computer Science 5502, p. 80-94 (2009).

[13] Lutz Schröder, Till Mossakowski. Generic Exception Handling and the Java Monad. AMAST 2004. Springer-Verlag Lecture Notes in Computer Science 3116, p. 443-459 (2004).

[14] Ross Tate. The sequential semantics of producer effect systems. POPL 2013. ACM Press, p. 15-26 (2013).

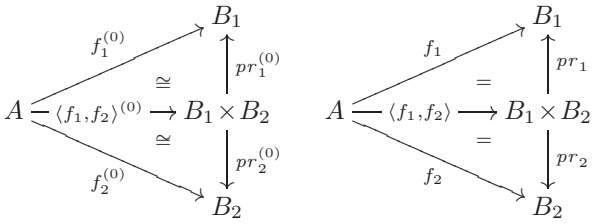
[15] Tarmo Uustalu, Varmo Vene. Comonadic Notions of Computation. CMCS 2008. ENTCS 203, p. 263-284 (2008).

[16] Philip Wadler. The essence of functional programming. POPL 1992. ACM Press, p. 1-14 (1992).

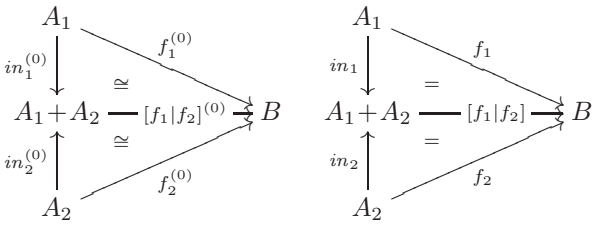
A Diagrams for decorated products and coproducts

ANY MONAD

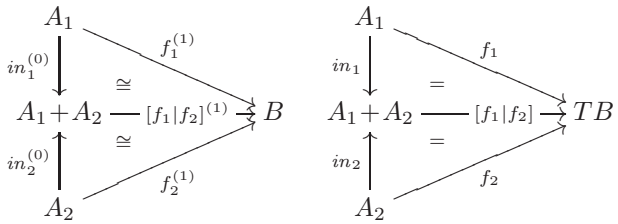
Products.



Coproducts.



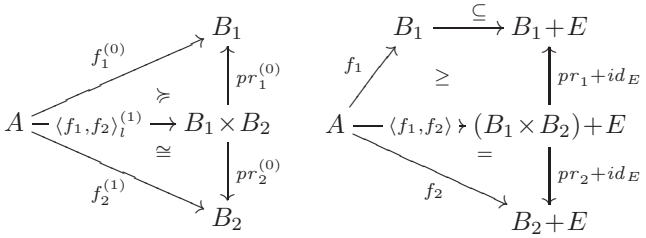
E.g., exceptions: for case distinction in the programmer's language:



THE MONAD OF EXCEPTIONS

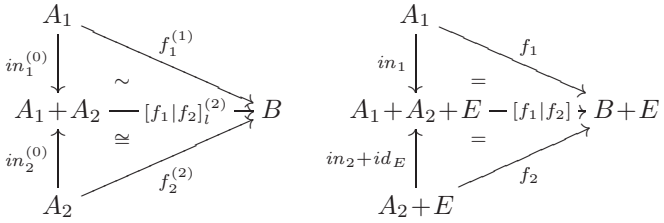
Products.

With $\preccurlyeq$. For sequential products in the programmer's language:



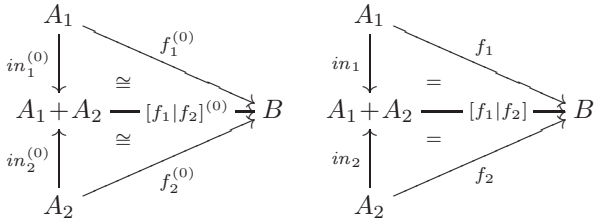
Coproducts.

Because + is AC. For case distinction in the core language:

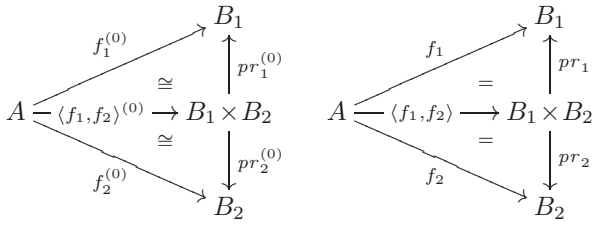


ANY COMONAD

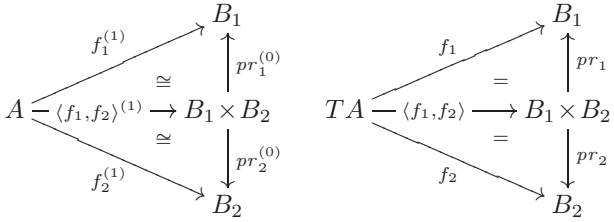
Coproducts.



Products.



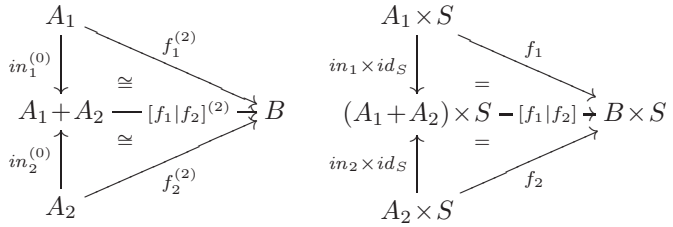
E.g., states: for pairs of observers:



THE COMONAD OF STATES

Coproducts.

When **C** is distributive. For case distinction:



Products.

Because $\times$ is AC. For sequential products: