HAL open science

Decorated proofs for computational effects: Exceptions

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

▶ To cite this version:

Jean-Guillaume Dumas, Dominique Duval, Laurent Fousse, Jean-Claude Reynaud. Decorated proofs for computational effects: Exceptions. [Research Report] Université Grenoble Alpes (UGA). 2012. hal-00678738

HAL Id: hal-00678738

https://hal.science/hal-00678738v1

Submitted on 13 Mar 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Decorated proofs for computational effects: Exceptions

Jean-Guillaume Dumas[*], Dominique Duval[†], Laurent Fousse[‡], Jean-Claude Reynaud[§]

March 13., 2012

Abstract

We define a proof system for exceptions which is close to the syntax for exceptions, in the sense that the exceptions do not appear explicitly in the type of any expression. This proof system is sound with respect to the intended denotational semantics of exceptions. With this inference system we prove several properties of exceptions. **Keywords.** Computational effects. Semantics of exceptions. Proof system.

Introduction

In this paper, as in the apparented papers [1, 4, 3], we consider that a *computational effect* in a language corresponds to an apparent lack of soundness: the intended denotational semantics is not a model of the syntax, but it becomes so when the syntax is endowed with relevant *decorations*; more precisely, a proof system can be designed for dealing with these decorations, which is sound with respect to the intended denotational semantics. In [3] this point of view has been applied to the side-effects due to the evolution of the *states* of the memory in an imperative or object-oriented language. In this paper, it is applied to the effects caused by *exceptions*. It happens that there is a duality between the denotational semantics of states and the *core part* of the semantics of exceptions [2]. The encapsulation of the core part inside the mechanism of exceptions is a succession of case distinctions; the proof system is extended for dealing with it. Properties of exceptions can be proved using this inference system and the proofs can be simplified by re-using proofs on states, thanks to the duality.

To our knowledge, the first categorical treatment of computational effects is due to Moggi [11]; this approach relies on monads, it is implemented in the programming language Haskell [16, 8]. Although monads are not used in this paper, the basic ideas underlying our approach rely on Moggi's remarks about notions of computations and monads. The examples proposed by Moggi include the exceptions monad $TA = A + E$ where E is the set of exceptions. Later on, using the correspondence between monads and algebraic theories, Plotkin and Power proposed to use Lawvere theories for dealing with the operations and equations related to computational effects [12, 9]; an operation is called *algebraic* when it satisfies some relevant genericity properties. The operation for raising exceptions is algebraic, while the operation for handling exceptions is not [13]. It follows that the handling of exceptions is quite difficult to formalize in this framework; several solutions are proposed in [15, 10, 14]. In this paper we rather use the categorical approach of *diagrammatic logics*, as introduced in [5] and developed in [1].

In Section 1 a denotational semantics for exceptions is defined, where we dissociate the core operations from their encapsulation. Then a decorated proof system and a decorated specification for exceptions are defined in Section 2 and it is checked that the denotational semantics for exceptions can be seen as a model of this specification. In Section 3 we use this framework for proving some properties of exceptions.

[*]LJK, Université de Grenoble, France. `Jean-Guillaume.Dumas@imag.fr`. This work is partly funded by the project HPAC of the French Agence Nationale de la Recherche (ANR 11 BS02 013).

[†]LJK, Université de Grenoble, France. `Dominique.Duval@imag.fr`. This work is partly funded by the project CLIMT of the French Agence Nationale de la Recherche (ANR 11 BS02 016).

[‡]LJK, Université de Grenoble, France. `Laurent.Fousse@imag.fr`

[§]Malhivert, Claix, France. `Jean-Claude.Reynaud@imag.fr`

1 Denotational semantics for exceptions

In this Section we define a denotational semantics of exceptions which relies on the semantics of exceptions in various languages, for instance in Java [6] and ML [7]. Syntax is introduced in Section 1.1 and the distinction between ordinary and exceptional values is discussed in Section 1.2. Denotational semantics of raising and handling exceptions are considered in Sections 1.3 and 1.4, respectively.

1.1 Signature for exceptions

The syntax for exceptions in computer languages depends on the language: the keywords for raising exceptions may be either `raise` or `throw`, and for handling exceptions they may be either `handle` or `try-catch`, for instance. In this paper we rather use `throw` and `try-catch`, but this choice does not matter. More precisely, the syntax of our language may be described in two parts: a *pure* part and an *exceptional* part. The pure part is a signature $\mathrm{Sig}_{\mathrm{pure}}$, made of types and operations; the $\mathrm{Sig}_{\mathrm{pure}}$-expressions are called the *pure expressions*. The interpretation of the pure expressions should neither raise nor handle exceptions. We assume that the pure operations are either constants or unary. General n-ary operations would require the use of sequential products, as in [4]; in order to focus on the fundamental properties of exceptions they are not considered in this paper. The exceptional part is made of a symbol E_i for each index i in some set of indices I, which is declared as: *Exception* E_i *of* P_i, where P_i is a pure type called the *type of parameters* for the *exceptional type* E_i (the P_i's need not be distinct). The exceptional types E_i provide familiar notations for the raising and handling operations and in Section 1.3 they are interpreted as sets, however we will not define any expression of type E_i.

Let us assume that the signature $\mathrm{Sig}_{\mathrm{pure}}$ is fixed. The *expressions* of our language are defined recursively from the pure operations and from the *raising* and *handling* operations, as follows.

Definition 1.1. Given a set of indices I and a symbol E_i for each $i \in I$, the *signature for exceptions* $\mathrm{Sig}_{\mathrm{exc}}$ is made of $\mathrm{Sig}_{\mathrm{pure}}$ together with a *raising* operation for each i in I and each type Y in $\mathrm{Sig}_{\mathrm{pure}}$:

$$\mathit{throw}_Y\, E_i \ : P_i \to Y \ .$$

and a *handling* operation for each $\mathrm{Sig}_{\mathrm{exc}}$-expression $f : X \to Y$, each non-empty list of indices $(i_1, \ldots, i_n)$ and each $\mathrm{Sig}_{\mathrm{exc}}$-expressions $g_1 : P_{i_1} \to Y, \ldots, g_n : P_{i_n} \to Y$:

$$\mathit{try}\{f\}\ \mathit{catch}\,\{E_{i_1} \Rightarrow g_1 | \ldots | E_{i_n} \Rightarrow g_n\} : X \to Y \ .$$

1.2 Ordinary values and exceptional values

The syntax for exceptions defined in Section 1.1 is now interpreted in the category of sets. In order to express the denotational semantics of exceptions, a major point is that there are two kinds of values: the ordinary (or non-exceptional) values and the exceptions. It follows that the operations may be classified according to the way they may, or may not, interchange these two kinds of values: an ordinary value may be *tagged* for constructing an exception, and later on the tag may be cleared in order to recover the value. Then we say that the exception gets *untagged*. Let us introduce a set *Exc* called the *set of exceptions*. For each set X we consider the disjoint union $X + \mathit{Exc}$ with the inclusions $\mathit{normal}_X : X \to X + \mathit{Exc}$ and $\mathit{abrupt}_X : \mathit{Exc} \to X + \mathit{Exc}$.

Definition 1.2. For each set X, an element of $X + \mathit{Exc}$ is an *ordinary value* if it is in $\mathit{normal}_X(X)$ and an *exceptional value* if it is in $\mathit{abrupt}_X(\mathit{Exc})$. A function $f : X + \mathit{Exc} \to Y + \mathit{Exc}$ is said to *raise an exception* if there is an element $x \in X$ such that $f(x) \in \mathit{Exc}$; *propagate exceptions* if $f(\mathit{abrupt}_X(e)) = \mathit{abrupt}_Y(e)$ for every $e \in \mathit{Exc}$; *recover from an exception* if there is some $e \in \mathit{Exc}$ such that $f(e) \in Y$.

We will use the same notations for the syntax and for its interpretation. Each type X is interpreted as a set X. Each pure expression $f_0 : X \to Y$ is interpreted as a function $f_0 : X \to Y$, which can be extended as $f = \mathit{normal}_Y \circ f_0 : X \to Y + \mathit{Exc}$. When $f : X \to Y$ is a $\mathrm{Sig}_{\mathrm{exc}}$-expression, which may involve some raising or

handling operation, its interpretation is a function $f : X \to Y + Exc$ which is defined in the next Sections 1.3 and 1.4. In addition, every function $f : X \to Y + Exc$ can be extended as $[f \mid abrupt_Y] : X + Exc \to Y + Exc$, which is defined by the equalities $[f \mid abrupt_Y] \circ normal_X = f$ and $[f \mid abrupt_Y] \circ abrupt_X = abrupt_Y$. This is the unique extension of f to $X + Exc$ which propagates exceptions.

Remark 1.3. *The interpretation of a* $\mathrm{Sig}_{\mathrm{exc}}$*-expression* $f : X \to Y$ *is a function which propagates exceptions; this function may raise exceptions but it cannot recover from an exception.* In Section 1.4, in order to catch exceptions, we will introduce functions which recover from exceptions. However such a function cannot be the interpretation of any $\mathrm{Sig}_{\mathrm{exc}}$-expression. Indeed, a *try-catch* expression may recover from exceptions which are raised inside the *try* block, but if an exception is raised before the *try-catch* expression is evaluated, this exception is propagated. Recovering from an exception can only be done by functions which are not expressible in the language generated by $\mathrm{Sig}_{\mathrm{exc}}$: such functions are called the *untagging* functions, they are defined in Section 1.4. Together with the *tagging* functions defined in Section 1.3 they are called the *core* functions for exceptions.

1.3 Tagging and raising exceptions: *throw*

Raising an exception is based on a tagging process, modelled as follows.

Definition 1.4. For each index $i \in I$ there is an injective function $t_i : P_i \to Exc$, called the *exception constructor* or the *tagging* function of index i, and the tagging functions for distinct indices have disjoint images. The image of t_i in Exc is denoted E_i.

Thus, the tagging function $t_i : P_i \to Exc$ maps a non-exceptional value (or *parameter*) $a \in P_i$ to an exception $t_i(a) \in Exc$. This means that the non-exceptional value a in P_i gets tagged as an exception $t_i(a)$ in Exc. The disjoint union of the E_i's is a subset of Exc; for simplicity we assume that $Exc = \sum_{i \in I} E_i$.

Definition 1.5. For each index $i \in I$ and each set Y, the *throwing* or *raising* function $throw_Y\, E_i$ is the tagging function t_i followed by the inclusion of Exc in $Y + Exc$: $throw_Y\, E_i = abrupt_Y \circ t_i : P_i \to Y + Exc$.

1.4 Untagging and handling exceptions: *try-catch*

Handling an exception is based on an untagging process for clearing the exception tags, which is modelled as follows.

Definition 1.6. For each index $i \in I$ there is a function $c_i : Exc \to P_i + Exc$, called the *exception recovery* or the *untagging* function of index i, which satisfies: $\forall a \in P_i\ c_i(t_i(a)) = a$ and $\forall b \in P_j\ c_i(t_j(b)) = t_j(b)$ for each $j \neq i$.

Thus, for each $e \in Exc$ the untagging function $c_i(e)$ tests whether the given exception e is in E_i; if this is the case, then it returns the parameter $a \in P_i$ such that $e = t_i(a)$, otherwise it propagates the exception e. Since it has been assumed that $Exc = \sum_{j \in I} E_j$, the untagging function $c_i(e)$ is uniquely determined by the above equalities.

For handling exceptions of type E_i raised by some function $f : X \to Y + Exc$, for i in a non-empty list $(i_1, \ldots, i_n)$ of indices, one provides for each k in $\{1, \ldots, n\}$ a function $g_k : P_{i_k} \to Y + Exc$ (which thus may itself raise exceptions). Then the handling process builds a function which encapsulates some untagging functions and which propagates exceptions. The indices $i_1, \ldots, i_n$ form a list: they are given in this order and they need not be pairwise distinct. It is assumed that this list is non-empty, because it is the usual choice in programming languages, however it would be easy to drop this assumption.

Definition 1.7. For each function $f : X \to Y + Exc$, each non-empty list $(i_1, \ldots, i_n)$ of indices in I and each family of functions $g_k : P_{i_k} \to Y + Exc$ (for $k \in \{1, \ldots, n\}$), the *handling* function

$$try\{f\}\ catch\,\{E_{i_1} \Rightarrow g_1 | \ldots | E_{i_n} \Rightarrow g_n\} : X \to Y + Exc$$

is defined as follows. Let $h = try\{f\}\ catch\,\{E_{i_1} \Rightarrow g_1 | \ldots | E_{i_n} \Rightarrow g_n\}$, for short. For each $x \in X$, $h(x) \in Y + Exc$ is defined in the following way.

First $f(x)$ is computed:
let $y = f(x) \in Y + Exc$.

(1) *If y is not an exception, then it is the required result:*
if $y \in Y$ then $h(x) = y \in Y \subseteq Y + Exc$.

(2) *If y is an exception, then:*

(a) *If the type of y is E_i for some index i in $(i_1, \ldots, i_n)$, then y has to be caught according to the first occurrence of the index i in the list:*
for each $k = 1, \ldots, n$,

- *Check whether the exception y has type E_{i_k}:*
  let $z = c_{i_k}(y) \in P_{i_k} + Exc$.
- *If the exception y has type E_{i_k} then it is caught:*
  if $z \in P_{i_k}$ then $h(x) = g_k(z) \in Y + Exc$.

(b) *If the type of y is E_i for some $i \notin \{i_1, \ldots, i_n\}$, then y is propagated:*
otherwise $h(x) = y \in Exc \subseteq Y + Exc$.

Equivalently, the definition of $h = try\{f\}\ catch\,\{E_{i_1} \Rightarrow g_1 | \ldots | E_{i_n} \Rightarrow g_n\}$ can be expressed as follows.

(1-2) The function $h : X \to Y + Exc$ is defined from f and from a function $catch\,\{E_{i_1} \Rightarrow g_1 | \ldots | E_{i_n} \Rightarrow g_n\} : Exc \to Y + Exc$ by:

$$h \;=\; \Big[\; normal_Y \mid catch\,\{E_{i_1} \Rightarrow g_1 | \ldots | E_{i_n} \Rightarrow g_n\} \;\Big] \circ f \tag{1}$$

$$\begin{tikzcd}
 & Y \arrow[d, "normal"] \arrow[drr, "normal"] & & \\
X \arrow[r, "f"] & Y + Exc \arrow[rr, "h"] & & Y + Exc \\
 & Exc \arrow[u, "abrupt"] \arrow[urr, "catch\,\{E_{i_1} \Rightarrow g_1 | \ldots | E_{i_n} \Rightarrow g_n\}"'] & &
\end{tikzcd}$$

(a-b) The function $catch\,\{E_{i_1} \Rightarrow g_1 | \ldots | E_{i_n} \Rightarrow g_n\}$ is obtained by setting $p = 1$ in the family of functions $k_p = catch\,\{E_{i_p} \Rightarrow g_p | \ldots | E_{i_n} \Rightarrow g_n\} : Exc \to Y + Exc$ (for $p = 1, \ldots, n$) which are defined recursively by:

$$k_p \;=\; \begin{cases} [\; g_n \mid abrupt_Y \;] \circ c_{i_n} & \text{when } p = n \\ [\; g_p \mid k_{p+1} \;] \circ c_{i_p} & \text{when } p < n \end{cases} \tag{2}$$

Let $k_{n+1} = abrupt_Y$, then $k_p \;=\; [\; g_p \mid k_{p+1} \;] \circ c_{i_p}$ for each $p \leq n$.

$$\begin{tikzcd}
 & P_{i_p} \arrow[d, "normal"] \arrow[drr, "g_p"] & & \\
Exc \arrow[r, "c_{i_p}"] & P_{i_p} + Exc \arrow[rr, "{[g_p | k_{p+1}]}"] & & Y + Exc \\
 & Exc \arrow[u, "abrupt"] \arrow[urr, "k_{p+1}"'] & &
\end{tikzcd}$$

When $n = 1$ we get $try\{f\}\ catch\,\{E_i \Rightarrow g\} = \Big[normal_Y | \big[g | abrupt_Y\big] \circ c_i\Big] \circ f$.

Remark 1.8. The handling process involves several nested case distinctions. Since it propagates exceptions, there is a first case distinction for checking whether the argument x is an exception (which is simply propagated) or not. If x is not an exception, then there is a case distinction (1-2) for checking whether $f(x)$ is an exception or not. If $f(x)$ is an exception then each step (a-b) checks whether the result of the untagging function is an exception. All these case distinctions check whether some value is an exception or not, they rely on disjoint unions of the form $T + Exc$. In contrast, for each step (a-b) there is another case distinction encapsulated in the computation of the untagging function, which checks whether the exception has the required exception type and relies on the disjoint union $Exc = \sum_i E_i$.

2 Decorated logic for exceptions

In Section 1 we have introduced a signature $\mathrm{Sig}_{\mathrm{exc}}$ and a denotational semantics for exceptions. However the soundness property is not satisfied: the denotational semantics is not a model of the signature, in the usual sense, since an expression $f : X \to Y$ is interpreted as a function $f : X \to Y + Exc$ instead of $f : X \to Y$. Therefore, in this Section we build a *decorated specification* for exceptions, including a "decorated" signature and "decorated" equations, which is sound with respect to the denotational semantics of Section 1. For this purpose, first we form an equational specification by extending the signature $\mathrm{Sig}_{\mathrm{exc}}$ with operations t_i and c_i and equations involving them, in order to formalize the tagging and untagging functions of Sections 1.3 and 1.4. Then we add *decorations* to this specification, and we define the interpretation of the expressions and equations according to their decorations. This means that we have to extend the equational logic with a notion of decoration; the decorations and the decorated inference rules are given in Section 2.1. In Section 2.2 we define the decorated specification for exceptions and in Section 2.3 we check that this decorated specification is sound with respect to the denotational semantics of Section 1. In the decorated specification for exceptions, there are on one side *private* operations for tagging and untagging exceptions, which do not appear in the signature for exceptions $\mathrm{Sig}_{\mathrm{exc}}$, and on the other side *public* operations for raising and handling exceptions, which are defined using the private operations. According to remark 1.3, an important feature of exceptions is that *all public operations propagate exceptions*, such operations will be called *propagators*; operations for recovering from exceptions may appear only as private operations, which will be called *catchers*.

2.1 Decorations

In order to deal with exceptions we define three decorations for expressions. They are denoted by (0), (1) and (2) used as superscripts, and their meaning is described in an informal way as follows.

- The interpretation of a *pure* expression $f^{(0)}$ may neither raise exceptions nor recover form exceptions.
- The interpretation of a *propagator* $f^{(1)}$ may raise exceptions but it is not allowed to recover from exceptions; thus, it must propagate all exceptions.
- The interpretation of a *catcher* $f^{(2)}$ may raise exceptions and recover form exceptions.

Every pure expression can be seen as a propagator and every propagator as a catcher. It follows that every expression can be seen as a catcher, so that the decoration (2) could be avoided; however we often use it for clarity.

In addition, we define two decorations for equations. They are denoted by two distinct relational symbols $\equiv$ for *strong* equations and by $\sim$ for *weak* equations. Using the fact that every expression can be seen as a catcher, their meaning can be described as follows.

- A *strong* equation $f^{(2)} \equiv g^{(2)}$ is interpreted as an equality of the functions f and g both on ordinary and on exceptional values.
- A *weak* equation $f^{(2)} \sim g^{(2)}$ is interpreted as an equality of the functions f and g on ordinary values, but f and g may differ on exceptional values.

Clearly every strong equation $f \equiv g$ gives rise to the weak equation $f \sim g$. On the other hand, since propagators cannot modifiy the exceptional values, every weak equation between propagators can be seen as a strong equation, and a similar remark holds for pure expressions.

Remark 2.1. It follows from these descriptions that every catcher k gives rise to a propagator $\triangledown k$ with a weak equation $k \sim \triangledown k$: this propagator $\triangledown k$ has the same interpretation as k on the non-exceptional values and it is interpreted as the identity on the exceptional values.

In the short note [2] it is checked that, from a denotational point of view, the functions for tagging and untagging exceptions are respectively *dual*, in the categorical sense, to the functions for looking up and updating states. It happens that this duality also holds from the decorated point of view. Thus, most of the decorated rules for exceptions are dual to the decorated rules for states [3]. The decorated rules for exceptions are given here in three parts (Figures 1, 2 and 3). For readability, the decoration properties are often grouped with other properties: for instance, "$f^{(1)} \sim g^{(1)}$" means "$f^{(1)}$ and $g^{(1)}$ and $f \sim g$".

The rules in Figure 1 may be called the rules for the *decorated monadic equational logic* for exceptions. The unique difference between these rules and the dual rules for states lies in the congruence rules for the weak equations: for states the replacement rule is restricted to pure g's, while for exceptions it is the substitution rule which is restricted to pure f's.

$$\frac{X}{id_X : X \to X} \qquad \frac{X}{id_X^{(0)}} \qquad \frac{f^{(0)}}{f^{(1)}}$$

$$\frac{f : X \to Y \quad g : Y \to Z}{g \circ f : X \to Z} \qquad \frac{f^{(0)} \quad g^{(0)}}{(g \circ f)^{(0)}} \qquad \frac{f^{(1)} \quad g^{(1)}}{(g \circ f)^{(1)}}$$

$$\frac{f : X \to Y}{f \circ id_X \equiv f} \qquad \frac{f : X \to Y}{id_Y \circ f \equiv f} \qquad \frac{f : X \to Y \quad g : Y \to Z \quad h : Z \to W}{h \circ (g \circ f) \equiv (h \circ g) \circ f}$$

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

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

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

$$\frac{f : X \to Y \quad g_1 \equiv g_2 : Y \to Z}{g_1 \circ f \equiv g_2 \circ f} \qquad \frac{f_1 \equiv f_2 : X \to Y \quad g : Y \to Z}{g \circ f_1 \equiv g \circ f_2}$$

$$\frac{f^{(0)} : X \to Y \quad g_1 \sim g_2 : Y \to Z}{g_1 \circ f \sim g_2 \circ f} \qquad \frac{f_1 \sim f_2 : X \to Y \quad g : Y \to Z}{g \circ f_1 \sim g \circ f_2}$$

Figure 1: Decorated rules for exceptions (1)

Several kinds of decorated coproducts are used for dealing with exceptions. The rules in Figure 2 are the rules for a decorated initial type $\mathbb{0}$, also called an *empty type*, and for a *constitutive coproduct*, as defined below. These rules are dual to the rules for the decorated final type and for the observational product for states in [3].

Definition 2.2. A *decorated initial type* for exceptions is a type $\mathbb{0}$ such that for every type X there is a pure expression $[\,]_X : \mathbb{0} \to X$ such that every function from $\mathbb{0}$ to X is weakly equivalent to $[\,]_X$.

It follows that every pure expression and every propagator from $\mathbb{0}$ to X is strongly equivalent to $[\,]_X$.

Definition 2.3. A *constitutive coproduct* for exceptions is a family of propagators $(q_i : X_i \to X)_i$ such that for every family of propagators $(f_i : X_i \to Y)_i$ there is a catcher $f = [f_i]_i : X \to Y$, unique up to strong equations, such that $f \circ q_i \sim f_i$ for each i.

This definition means that a constitutive coproduct can be used for building a catcher from several propagators; this corresponds to the fact that the set *Exc* is the disjoint union of the E_i's.

The next property corresponds to remark 2.1.

Definition 2.4. For each catcher $k^{(2)} : X \to Y$ there is a propagator $\triangledown k^{(1)} : X \to Y$, unique up to strong equations, such that $\triangledown k^{(1)} \sim k^{(2)}$.

When $\mathbb{0}$ is a *decorated initial type:*

$$\frac{X}{[\,]_X : \mathbb{0} \to X} \qquad \frac{X}{[\,]_X^{(0)}} \qquad \frac{f : \mathbb{0} \to Y}{f \sim [\,]_Y}$$

When $(q_i^{(1)} : X_i \to X)_i$ is a *constitutive coproduct:*

$$\frac{(f_i^{(1)} : X_i \to Y)_i}{[f_i]_i^{(2)} : X \to Y} \qquad \frac{(f_i^{(1)} : X_i \to Y)_i}{[f_j]_j \circ q_i \sim f_i}$$

$$\frac{(f_i^{(1)} : X_i \to Y)_i \quad f^{(2)} : X \to Y \quad \forall i\ f \circ q_i \sim f_i}{f \equiv [f_j]_j}$$

Figure 2: Decorated rules for exceptions (2)

According to the previous rules, for each type X there are two pure expressions $id_X : X \to X$ and $[\,]_X : \mathbb{0} \to X$. It is straightforward to check that they form a coproduct with respect to pure expressions and strong equations: for each $f^{(0)} : X \to Y$ and $g^{(0)} : \mathbb{0} \to Y$ there is a pure expression $[f|g]^{(0)} : X \to Y$, unique up to strong equations, such that $[f|g] \circ id_X \equiv f$ and $[f|g] \circ [\,]_X \equiv g$, indeed such a situation implies that $g \equiv [\,]_X$ and $[f|g] \equiv f$. This pure coproduct, with coprojections id_X and $[\,]_X$, is called *the coproduct* $X \cong X + \mathbb{0}$. In addition, we assume that it satisfies the following *decorated coproduct* property.

Definition 2.5. For each propagator $g^{(1)} : X \to Y$ and each catcher $k^{(2)} : \mathbb{0} \to Y$ there is a catcher $[g\,|\,k]^{(2)} : X \to Y$, unique up to strong equations, such that $[g\,|\,k]^{(2)} \sim g^{(1)}$ and $[g\,|\,k]^{(2)} \circ [\,]_X^{(0)} \equiv k^{(2)}$.

The rules in Figure 3 are the rules for the construction of ∇k and for the decorated coproduct $X \cong X + \mathbb{0}$. They will be used for building the handling operations from the untagging operations.

$$\frac{k^{(2)} : X \to Y}{\nabla k^{(1)} : X \to Y} \qquad \frac{k^{(2)} : X \to Y}{\nabla k \sim k}$$

$$\frac{g^{(1)} : X \to Y \quad k^{(2)} : \mathbb{0} \to Y}{[g\,|\,k]^{(2)} : X \to Y} \qquad \frac{g^{(1)} : X \to Y \quad k^{(2)} : \mathbb{0} \to Y}{[g\,|\,k] \sim g} \qquad \frac{g^{(1)} : X \to Y \quad k^{(2)} : \mathbb{0} \to Y}{[g\,|\,k] \circ [\,]_X \equiv k}$$

$$\frac{g^{(1)} : X \to Y \quad k^{(2)} : \mathbb{0} \to Y \quad f^{(2)} : X \to Y \quad f \sim g \quad f \circ [\,]_X \equiv k}{f \equiv [g\,|\,k]}$$

Figure 3: Decorated rules for exceptions (3)

2.2 A decorated specification for exceptions

Let $\mathcal{L}$ denote the inference system provided by the decorated rules for exceptions (Figures 1, 2 and 3). As for other inference systems, we may define *theories* and *specifications* (or *presentations of theories*) with respect to $\mathcal{L}$. They are called *decorated specifications* and *decorated theories*, respectively. This approach is based on the general framework for *diagrammatic* theories and specifications [5, 1], but no knowledge of this framework is assumed in this paper. A decorated theory is made of types, expressions, equations and coproducts which satisfy the decorated rules for exceptions. In this Section we define a decorated specification Σ_{exc}, which may be used for generating a decorated theory by applying the decorated inference rules for exceptions.

Definition 2.6. Let Σ_{pure} be some fixed equational specification (as in Section 1.1 for simplicity it is assumed that Σ_{pure} has no n-ary operation with $n > 1$). The *decorated specification for exceptions* Σ_{exc} is made of the equational specification Σ_{pure} where each operation is decorated as pure and each equation as strong (or weak, since both coincide here), a type $\mathbb{0}$ called the *empty* type and for each i in some set I a type P_i (of *parameters*) in Σ_{pure}, a propagator $t_i^{(1)} : P_i \to \mathbb{0}$, a catcher $c_i^{(2)} : \mathbb{0} \to P_i$ and the weak equations: $c_i \circ t_i \sim id : P_i \to P_i$ and $c_i \circ t_j \sim [\,] \circ t_j : P_j \to P_i$ for every $j \in I,\ j \neq i$.

Definition 2.7. For each i in I and each type Y in Σ_{pure} the *raising* propagator

$$(\mathit{throw}_Y\, E_i\,)^{(1)} : P_i \to Y$$

is defined as

$$\mathit{throw}_Y\, E_i \;= [\,]_Y \circ t_i : P_i \to Y\ .$$

According to remark 1.3, the handling operation $\mathit{try}\{f\}\ \mathit{catch}\{\dots\}$ is a propagator, not a catcher: indeed, it may recover from exceptions which are raised by f, but it must propagate exceptions which are raised before $\mathit{try}\{f\}\ \mathit{catch}\{\dots\}$ is called.

Definition 2.8. For each propagator $f^{(1)} : X \to Y$, each non-empty list of indices $(i_1, \dots, i_n)$ and each propagators $g_1^{(1)} : P_{i_1} \to Y, ..., g_n^{(1)} : P_{i_n} \to Y$, the *handling* propagator

$$(\mathit{try}\{f\}\ \mathit{catch}\,\{E_{i_1} \Rightarrow g_1 | \dots | E_{i_n} \Rightarrow g_n\})^{(1)} : X \to Y$$

is defined as follows.

(A-B) The propagator $\mathit{try}\{f\}\ \mathit{catch}\,\{E_{i_1} \Rightarrow g_1 | \dots | E_{i_n} \Rightarrow g_n\} : X \to Y$ is defined from a catcher $H : X \to Y$ by:

$$(\mathit{try}\{f\}\ \mathit{catch}\,\{E_{i_1} \Rightarrow g_1 | \dots | E_{i_n} \Rightarrow g_n\})^{(1)} \;=\; (\triangledown H)^{(1)} : X \to Y$$

(1-2) The catcher $H : X \to Y$ is defined from the propagator $f : X \to Y$ and from a catcher $k_1 = \mathit{catch}\,\{E_{i_1} \Rightarrow g_1 | \dots | E_{i_n} \Rightarrow g_n\} : \mathbb{0} \to Y$ by:

$$H^{(2)} \;=\; \left[id_Y^{(0)} \mid k_1^{(2)} \right]^{(2)} \circ f^{(1)} : X \to Y$$

$$\begin{tikzcd}
 & Y \arrow[d, "id"'] \arrow[drr, "id"] \arrow[drrr, phantom, "\sim"] & & \\
X \arrow[r, "f"] & Y \arrow[rrr, "h"] \arrow[drrr, phantom, "\equiv"] & & Y \\
 & \mathbb{0} \arrow[u, "{[\,]}"] \arrow[urr, "k_1"'] & &
\end{tikzcd}$$

(a-b) The catcher $k_1 : \mathbb{0} \to Y$ is obtained by setting $p = 1$ in the family of catchers $k_p = \mathit{catch}\,\{E_{i_p} \Rightarrow g_p | \dots | E_{i_n} \Rightarrow g_n\} : \mathbb{0} \to Y$ (for $p = 1, \dots, n$) which are defined recursively by:

$$k_p \;=\; [g_p \mid k_{p+1}] \circ c_{i_p} \text{ for each } p = 1, \dots, n \;\text{ and }\; k_{n+1} = [\,]_Y$$

$$\begin{tikzcd}
 & P_{i_p} \arrow[d, "id"'] \arrow[drr, "g_p"] \arrow[drrr, phantom, "\sim"] & & \\
\mathbb{0} \arrow[r, "c_{i_p}"] & P_{i_p} \arrow[rrr, "{[g_p|k_{p+1}]}"] \arrow[drrr, phantom, "\equiv"] & & Y \\
 & \mathbb{0} \arrow[u, "{[\,]}"] \arrow[urr, "k_{p+1}"'] & &
\end{tikzcd}$$

It will be proved in Lemma 3.2 that since $k_{n+1} = [\,]_Y$ we have $[g_n | k_{n+1}] \equiv g_n$. It follows that when $n = 1$ and 2 we get respectively:

$$\mathit{try}\{f\}\ \mathit{catch}\,\{E_i \Rightarrow g\} \;\equiv\; \triangledown\big(\ \big[id_Y \mid g \circ c_i\big] \circ f\ \big) \tag{3}$$

$$\mathit{try}\{f\}\ \mathit{catch}\,\{E_i \Rightarrow g \mid E_j \Rightarrow h\} \;\equiv\; \triangledown\big(\ \big[id \mid [g \mid h \circ c_j] \circ c_i\big] \circ f\ \big) \tag{4}$$

2.3 Decorated models

Let Exc be a set and P_i (for $i \in I$) a family of sets with injections $t_i : P_i \to Exc$, such that Exc is the disjoint union of the images $E_i = t_i(P_i)$. Then we may define a decorated theory Θ_{exc} as follows. A type is a set, a pure expression $f^{(0)} : X \to Y$ is a function $f : X \to Y$, a propagator $f^{(1)} : X \to Y$ is a function $f : X \to Y + Exc$ and a catcher $f^{(2)} : X \to Y$ is a function $f : X + Exc \to Y + Exc$. For instance, each injection $t_i : P_i \to Exc$ is a propagator $t_i^{(1)} : P_i \to \emptyset$. The conversion from pure expressions to propagators is the construction of $f_1 = normal_Y \circ f_0 : X \to Y + Exc$ from $f_0 : X \to Y$ and the conversion from propagators to catchers is the construction of $f_2 = [f \mid abrupt_Y] : X + Exc \to Y + Exc$ from $f_1 : X \to Y + Exc$. Composition of two expressions can be defined by converting them to catchers and using the composition of functions $f : X + Exc \to Y + Exc$ and $g : Y + Exc \to Z + Exc$ as $g \circ f : X + Exc \to Z + Exc$. When restricted to propagators this is compatible with the Kleisli composition with respect to the monad $X + Exc$. When restricted to pure expressions this is compatible with the composition of functions $f_0 : X \to Y$ and $g_0 : Y \to Z$ as $g_0 \circ f_0 : X \to Z$. A strong equation $f^{(2)} \equiv g^{(2)} : X \to Y$ is an equality $(\forall x \in X + Exc,\ f(x) = g(x))$, and a weak equation $f \sim g : X \to Y$ is an equality $(\forall x \in X,\ f(x) = g(x))$; when restricted to propagators both notions coincide. The empty set $\emptyset$ is a decorated initial type and the family of propagators $(t_i^{(1)} : P_i \to \emptyset)$ is a constitutive coproduct, because the family of functions $(t_i : P_i \to Exc)$ is a coproduct in the category of sets.

The *models* of a decorated specification Σ with values in a decorated theory Θ are defined as kinds of morphisms from Σ to Θ in [1]: a model maps each feature (type, pure expression, propagator, catcher, decorated initial type, constitutive coproduct, ...) of Σ to a feature of the same kind in Θ. When Θ is the theory Θ_{exc} we recover the meaning of decorations as given informally in Section 2.1. When in addition Σ is the specification Σ_{exc} we get the following result.

Theorem 2.9. *The decorated specification for exceptions Σ_{exc} is sound with respect to the denotational semantics of Section 1, in the sense that by mapping every feature in Σ_{exc} to the feature with the same name in the decorated theory Θ_{exc} we get a model of Σ_{exc} with values in Θ_{exc}.*

Proof. For the tagging and untagging operations this is clear from the notations. Then for the raising operations the result is obvious. For the handling operations the result comes from a comparison of the steps (1-2) and (a-b) in Definitions 2.8 and 1.7, while step (A-B) in Definition 2.8 corresponds to the propagation of exceptions by the handling functions, as in remark 1.8. □

3 Proofs involving exceptions

As for proofs on states in [3], we may consider two kinds of proofs on exceptions: the *explicit* proofs involve a type of exceptions, while the *decorated* proofs do not mention any type of exceptions but require the specification to be decorated, in the sense of Section 2. In addition, there is a simple procedure for deriving an explicit proof from a decorated one. In this Section we give some decorated proofs for exceptions, using the inference rules of Section 2.1. Since the properties of the core tagging and untagging operations are dual to the properties of the looking up and updating operations we may reuse the decorated proofs involving states from [3]. Starting from any one of the seven equations for states in [12] we can dualize this equation and derive a property about raising and handling exceptions. This is done in this Section for two of these equations.

On states, the *annihilation lookup-update* property means that updating any location with the content of this location does not modify the state. A decorated proof of this property is given in [3]. By duality we get the following *annihilation untag-tag* property, which means that tagging just after untagging, both with respect to the same exceptional type, returns the given exception.

Lemma 3.1 (Annihilation untag-tag)**.** *For each $i \in I$: $t_i^{(1)} \circ c_i^{(2)} \equiv id_{\emptyset}^{(0)}$.*

Lemma 3.1 is used in Proposition 3.3 for proving the *annihilation catch-raise* property: catching an exception by re-raising it is like doing nothing. First, let us prove Lemma 3.2, which has been used for getting Equation (3).

Lemma 3.2. *For each propagator $g^{(1)} : X \to Y$ we have $[g|\,[\,]_Y]^{(2)} \equiv g^{(1)}$.*

Proof. Since $[g|\,[\,]_Y]$ is characterized up to strong equations by $[g|\,[\,]_Y] \sim g$ and $[g|\,[\,]_Y] \circ [\,]_X \equiv [\,]_Y$, we have to prove that $g \sim g$ and $g \circ [\,]_X \equiv [\,]_Y$. The weak equation is due to the reflexivity of $\sim$. The unicity of $[\,]_Y$ up to weak equations implies that $g \circ [\,]_X \sim [\,]_Y$, and since both members are propagators we get $g \circ [\,]_X \equiv [\,]_Y$. □

Proposition 3.3 (Annihilation catch-raise)**.** *For each propagator $f^{(1)} : X \to Y$ and each $i \in I$: $try\{f\}\ catch\,\{E_i \Rightarrow throw_Y\ E_i\} \equiv f$.*

Proof. By Equation (3) and Definition 2.7 we have $try\{f\}\ catch\,\{E_i \Rightarrow throw_Y\ E_i\} \equiv \triangledown([id_Y|\,[\,]_Y \circ t_i \circ c_i] \circ f)$. By Lemma 3.1 $[id_Y|\,[\,]_Y \circ t_i \circ c_i] \equiv [id_Y|\,[\,]_Y]$, and the unicity property of $[id_Y|\,[\,]_Y]$ implies that $[id_Y|\,[\,]_Y] \equiv id_Y$. Thus $try\{f\}\ catch\,\{E_i \Rightarrow throw_Y\ E_i\} \equiv \triangledown f$. Finally, since $\triangledown f \sim f$ and f is a propagator we get $\triangledown f \equiv f$. □

On states, the *commutation update-update* property means that updating two different locations can be done in any order. By duality we get the following *commutation untag-untag* property, which means that untagging with respect to two distinct exceptional types can be done in any order. A detailed decorated proof of the commutation update-update property is given in [3]. The statement of this property and its proof use *semi-pure products*, which were introduced in [4] in order to provide a decorated alternative to the strength of a monad. Dually, the commutation untag-untag property use *semi-pure coproducts*, which generalize the decorated coproducts $X \cong X + \mathbb{0}$ from Definition 2.5. The *coproduct* of two types A and B is defined as a type $A + B$ with two pure coprojections $q_1^{(0)} : A \to A + B$ and $q_2^{(0)} : B \to A + B$, which satisfy the usual categorical coproduct property with respect to the pure morphisms. Then the *semi-pure coproduct* of a propagator $f^{(1)} : A \to C$ and a catcher $k^{(2)} : B \to C$ is a catcher $[f|k]^{(2)} : A + B \to C$ which is characterized, up to strong equations, by the following decorated version of the coproduct property: $[f|k] \circ q_1 \sim f$ and $[f|k] \circ q_2 \equiv k$. Then as usual, the coproduct $f' + k' : A + B \to C + D$ of a propagator $f' : A \to C$ and a catcher $k' : B \to D$ is the catcher $f' + k' = [q_1 \circ f \mid q_2 \circ k] : A + B \to C + D$. Whenever g is a propagator it can be proved that $\triangledown\,[f|g] \equiv [f|g]$; thus, up to strong equation, we can assume that in this case $[f \mid g] : A + B \to C$ is a propagator; it is characterized, up to strong equations, by $[f \mid g] \circ q_1 \equiv f$ and $[f \mid g] \circ q_2 \equiv g$.

Lemma 3.4 (Commutation untag-untag)**.** *For each $i, j \in I$ with $i \neq j$:*

$$(c_i + id_{P_j})^{(2)} \circ c_j^{(2)} \equiv (id_{P_i} + c_j)^{(2)} \circ c_i^{(2)} : \mathbb{0} \to P_i + P_j$$

Proposition 3.5 (Commutation catch-catch)**.** *For each $i, j \in I$ with $i \neq j$:*

$$try\{f\}\ catch\,\{E_i \Rightarrow g \mid E_j \Rightarrow h\} \equiv try\{f\}\ catch\,\{E_j \Rightarrow h \mid E_i \Rightarrow g\}$$

Proof. According to Equation (4): $try\{f\}\ catch\,\{E_i \Rightarrow g \mid E_j \Rightarrow h\} \equiv \triangledown([id \mid [g \mid h \circ c_j] \circ c_i] \circ f)$ Thus, the result will follow from $[g \mid h \circ c_j] \circ c_i \equiv [h \mid g \circ c_i] \circ c_j$. It is easy to check that $[g \mid h \circ c_j] \equiv [g \mid h] \circ (id_{P_i} + c_j)$, so that $[g \mid h \circ c_j] \circ c_i \equiv [g \mid h] \circ (id_{P_i} + c_j) \circ c_i$. Similarly $[h \mid g \circ c_i] \circ c_j \equiv [h \mid g] \circ (id_{P_j} + c_i) \circ c_j$ hence $[h \mid g \circ c_i] \circ c_j \equiv [g \mid h] \circ (c_i + id_{P_j}) \circ c_j$. Then the result follows from Lemma 3.4. □

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. A duality between exceptions and states. Accepted for publication in MSCS. arXiv:1112.2394.

[3] Jean-Guillaume Dumas, Dominique Duval, Laurent Fousse, Jean-Claude Reynaud. Decorated proofs for computational effects: States. Accepted for presentation at ACCAT'12. arXiv:1112.2396.

[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] Dominique Duval. Diagrammatic Specifications. Mathematical Structures in Computer Science 13, p. 857-890 (2003).

[6] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. The Java Language Specification, Third Edition. Addison-Wesley Longman (2005). docs.oracle.com/javase/specs/jls/se5.0/jls3.pdf.

[7] Robert Harper. Programming in Standard ML. www.cs.cmu.edu/ rwh/smlbook/book.pdf.

[8] The Haskell Programming Language. Monads. www.haskell.org/haskellwiki/Monad.

[9] Martin Hyland, John Power. The Category Theoretic Understanding of Universal Algebra: Lawvere Theories and Monads. Electronic Notes in Theoretical Computer Science 172, p. 437-458 (2007).

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

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

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

[13] Gordon D. Plotkin, John Power. Algebraic Operations and Generic Effects. Applied Categorical Structures 11(1), p. 69-94 (2003).

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

[15] 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).

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