A decorated proof system for exceptions

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

A decorated proof system for exceptions

Jean-Guillaume Dumas* Dominique Duval*

Jean-Claude Reynaud†

October 8, 2013

Abstract

In this paper, we first provide a careful description of the denotational semantics of exceptions in an object-oriented setting. Then we define a proof system for exceptions which is sound with respect to this denotational semantics. Our proof system is close to the syntax, as in effect systems, in the sense that the exceptions do not appear explicitly in the type of expressions which may raise them, much like compiler qualifiers or specifiers. But our system also involves different kind of equations, in order to separate the verification of properties that are true only up to effects from the verification of generic properties. Thanks to a duality between the global state effect and the core part of the exception effect, the proofs are in two parts: the first part is generic and can be directly dualized from the proofs on global states, while the second part uses specific rules. These specific rules are related to the encapsulation of the core part into some control for the conditional raising and handling of exceptions.

Keywords: Semantics of exceptions in an object-oriented setting. Proof system for exceptions. Computational effects.

Introduction

Exceptions form a *computational effect*, in the sense that a syntactic expression $f : X \to Y$ is not always interpreted as a function $[\![f]\!] : [\![X]\!] \to [\![Y]\!]$. For instance a function which raises an exception has to be interpreted as a function $[\![f]\!] : [\![X]\!] \to [\![Y]\!] + Exc$ where Exc is the set of exceptions and "+" denotes the disjoint union. In a programming language, exceptions usually differ from errors in the sense that it is possible to recover from an exception while this is impossible for an error; thus, exceptions have to be both raised and handled. There are exceptions in most object-oriented programming languages, and they are usually the objects of classes which are related by a hierarchy of subclasses.

*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.

In this paper we define the syntax of a simple language for dealing with exceptions which are organized in a hierarchy of subtypes. We add "decorations" to this syntax, in order to classify the expressions of the language according to their interaction with the exceptions. Decorations extend the syntax much like compiler qualifiers or specifiers. These decorations are similar to the *effect systems* for instance of [14] or [27], where each function can be labelled by an effect. Here, we also decorate the equations.

Moreover, we define an inference system for dealing with this decorated syntax, and we prove that this inference system is sound with respect to the semantics of exceptions. The result of this paper is then that we can use this new inference system for proving various properties of exceptions in a novel way. Indeed, we can separate the verification of the proofs in two steps: a first step checks properties of the programs, *up to effects*, while a second step takes the effect into account via the decorations.

Our method relies on a general algebraic framework for computational effects based on category theory [3]. This framework has been used for dealing with several issues related to effects [10, 4, 8]. This method has led to the discovery of a duality between global states and exceptions, in which catching an exception is dual to updating a state; this duality formalizes the fact that states are *observed* while exceptions are *constructed* [7].

In contrast with this categorical approach, in this paper we focus on a logical approach which is better suited to the construction of a proof system. Furthermore, with this point of view it should be easier to take advantage of the use of a proof assistant. In addition, we extend our framework in the direction of object-oriented languages by taking into account a hierarchy of types for exceptions. It should be noted that we distinguish the private operation of catching an exception from the public operation of handling it (also called "try/catch"), More precisely, in the mechanism for exceptions, we distinguish a *core part* from a *control part*. The core part is private, it is made of *tagging* and *untagging* operations, respectively dual from *lookup* and *update* functions for states. The tagging operations construct exceptions from a parameter while the untagging operations recover the parameter from an exception. Then, this core part is encapsulated inside the control part via a succession of case distinctions. Properties of exceptions in programming languages can be proved using this inference system, and the proofs usually have two parts: the first part can be obtained "for free" by dualizing a proof on global states while the second part is specific to exceptions.

To our knowledge, the first algebraic treatment of computational effects is due to Moggi [19]; this approach relies on *monads* and is implemented in the programming language Haskell [26, 16]. The examples proposed by Moggi include the global states monad $TX = (X \times St)^{St}$, where St is the set of states, and the exceptions monad $TX = X + Exc$, where Exc is the set of exceptions. Later on, Plotkin and Power proposed to use *Lawvere theories* for dealing with the operations and equations related to computational effects [20, 17]. With this approach, it is inherently different and more difficult to handle exceptions than to update states [21, 25, 18, 22]. Effect systems are related to monads,

see [27], and exceptions in Java, also studied from a coalgebraic point of view, can be found in [13]. We here rather use the extension of effect systems where equations are also decorated, but still in a categorical framework [10].

In Section 1, we make precise the syntax we use for dealing with exceptions in an object-oriented setting. Then we describe the intended denotational semantics of exceptions in Section 2, dissociating the core operations from their encapsulation. We propose in Section 3 a decorated inference system for exceptions, and we prove its soundness with respect to the intended semantics in Theorems 3.1 and 3.6. Appendix A is an illustration of the approach, where some properties of exceptions are proven using the decorated inference system.

1 Syntax

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`, `try-with`, `try-except` or `try-catch`, for instance. In this paper we rather use `throw` and `try-catch`. The syntax for dealing with exceptions may be described in two parts: a basic part which deals with the basic data types and an exceptional part for raising and handling exceptions.

The *basic* part of the syntax is a signature Sig_{base}, made of a *types* (or *sorts*) and *operations*. For simplicity we assume that the operations in Sig_{base} are either constants or unary; general n-ary operations will be mentioned in Section 3.5.

The signature Sig_{exc} for exceptions is made of Sig_{base} together with the operations for raising and handling exceptions. The *exceptional types* form a subset $\mathcal{T}$ of the set of types of Sig_{base}. For instance in C++ any type (basic type or class) is an exceptional type, while in Java, or more generally in [5], there is a base class for exceptional types, such that the exceptional types are precisely the subtypes of this base class. Moreover, in this paper we consider a simple *hierarchy* of types, as follows.

Definition 1.1. Given a signature Sig_{base}, a *hierarchy of exceptional types* on Sig_{base} is a partially ordered set $(\mathcal{T}, \twoheadrightarrow)$ made of a subset $\mathcal{T}$ of types of Sig_{base} and a partial order $\twoheadrightarrow$ on $\mathcal{T}$ called the *subtyping relation*. Given a hierarchy of exceptional types $(\mathcal{T}, \twoheadrightarrow)$ on Sig_{base}, the signature $Sig_{base,\twoheadrightarrow}$ is made of Sig_{base} together with, for each T' and T in $\mathcal{T}$ such that $T' \twoheadrightarrow T$, an operation $cast_{T',T} : T' \to T$ called the *cast* operation from T' to T.

A hierarchy is called *discrete* when the unique subtype of each type T is T itself. The duality between exceptions and states, as presented in [7], can be used for deriving properties of exceptions from properties of states under the assumption that the hierarchy of exceptional types is discrete.

The signature Sig_{exc} for exceptions is made of $Sig_{base,\twoheadrightarrow}$ together with the operations for raising and handling exceptions, as follows.

Definition 1.2. Let Sig_{base} be a signature and $(\mathcal{T}, \twoheadrightarrow)$ a hierarchy of exceptional types on Sig_{base}. The *signature for exceptions* Sig_{exc} is made of $Sig_{base,\twoheadrightarrow}$ together with, for each exceptional type T and each type Y in Sig_{base} a *raising* (or *throwing*) operation:

$$throw_{T,Y} : T \to Y \ ,$$

and a *handling* (or *try-catch*) operation for each Sig_{exc}-term $f : X \to Y$, each non-empty list of exceptional types $(T_1, \ldots, T_n)$ and each family of Sig_{exc}-terms $g_1 : T_1 \to Y$, $\ldots$, $g_n : T_n \to Y$:

$$try\{f\}\ catch\,\{T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n\} : X \to Y \ .$$

An important, and somewhat surprising, feature of a language with exceptions is that all expressions in the language, including the *try-catch* expressions, propagate exceptions. Indeed, if an exception is raised before some *try-catch* expression is evaluated, this exception is propagated. In fact, the *catch* block in a *try-catch* expression may recover from exceptions which are raised inside the *try* block, but the *catch* block alone is not an expression of the language.

More precisely, the operations for dealing with exceptions can be expressed in terms of more elementary operations, which may be seen as *private* operations of the language. The *tagging* operations will be used for raising exceptions, using a private *empty type* $\mathbb{0}$. The *untagging* operations will be used for handling exceptions, more precisely for catching them inside the *catch* block in any *try-catch* expression. We call them the *core operations* for exceptions. They are not part of Sig_{exc}, but the interpretation of the operations for raising and handling exceptions, which are part of Sig_{exc}, will be defined in terms of the interpretations of the core operations.

Definition 1.3. Let Sig_{exc} be a signature for exceptions. The *core* of Sig_{exc} is made of a type $\mathbb{0}$ called the *empty type* and two operations for each exceptional type T: an operation $tag_T : T \to \mathbb{0}$ called the *exception constructor* or the *tagging* operation for T and an operation $untag_T : \mathbb{0} \to T$ called the *exception recovery* or the *untagging* operation for T.

2 Denotational semantics

In this Section we define a denotational semantics of exceptions which relies on the semantics of exceptions in various languages, for instance in C++ [1, Ch. 15], Java [12, Ch. 14] and ML [15].

The basic part of the syntax is interpreted in the usual way: each type X is interpreted as a set $[\![X]\!]$ and each operation $f : X \to Y$ of Sig_{base} as a function $[\![f]\!] : [\![X]\!] \to [\![Y]\!]$. Each cast operation $cast_{T',T} : T' \to T$ is also interpreted as a function $[\![cast_{T',T}]\!] : [\![T']\!] \to [\![T]\!]$; the interpretations of the cast operations must be such that $[\![cast_{T,T}]\!]$ is the identity on T and when $T'' \twoheadrightarrow T' \twoheadrightarrow T$ then $[\![cast_{T'',T}]\!] = [\![cast_{T',T}]\!] \circ [\![cast_{T'',T'}]\!]$.

When $h : X \to Y$ in Sig_{exc} is a raising or handling operation, it is not interpreted as a function $[\![h]\!] : [\![X]\!] \to [\![Y]\!]$: this corresponds to the fact that the

exceptions form a *computational effect*. Let us begin with an informal description of $[\![h]\!]$. If $h = throw_{T,Y}$ then $[\![h]\!]$ signals an error, which may be "caught" by an exception handler; the function $[\![h]\!]$ turns the parameter of type T into an exception, in such a way that this exception is considered as being of type Y. If $h = try\{f\}\ catch\,\{T \Rightarrow g\}$ then $[\![h]\!](x)$ returns the same result as $[\![f]\!](x)$ when $[\![f]\!](x)$ does not raise any exception; if $[\![f]\!](x)$ raises an exception of type T' for some subtype T' of T then this exception is caught, which means that its parameter y is recovered and $[\![h]\!](x)$ returns the same result as $[\![g]\!](y)$ (this result may be an exception); otherwise, i.e., when $[\![f]\!](x)$ raises an exception of type T' where T' is not a subtype of T, the exception is returned (which usually produces an error message like "*uncaught exception...*"). The interpretation of $try\{f\}\ catch\,\{T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n\}$ for any $n > 1$ is similar; it is checked whether the exception returned by f has type T_1 or T_2 ... or T_n in this order, taking into account possible inheritance, so that whenever $T_k = T_j$, or more generally $T_k \twoheadrightarrow T_j$, with $j < k$, the clause $T_k \Rightarrow g_k$ is never executed.

The distinction between ordinary and exceptional values is discussed in Subsection 2.1. Then, denotational semantics of raising and handling exceptions are considered in Subsections 2.2 and 2.3, respectively. We assume that some interpretation of $Sig_{base,\twoheadrightarrow}$ has been chosen.

2.1 Ordinary values and exceptional values

In order to express the denotational semantics of exceptions, a major point is the distinction between 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*.

Definition 2.1. The *set of exceptions* Exc is the disjoint union of the sets $[\![T]\!]$ for all the exceptional types T.

Definition 2.2. For each set A, the set $A + Exc$ is the disjoint union of A and Exc and the canonical inclusions are denoted (when needed) $normal_A : A \to A + Exc$ and $abrupt_A : Exc \to A + Exc$. An element of $A + Exc$ is an *ordinary value* if it is in A and an *exceptional value* if it is in Exc.

In the denotational semantics for exceptions, we will see that an operation $f : X \to Y$ of Sig_{exc} may be interpreted either as a function $[\![f]\!] : [\![X]\!] \to [\![Y]\!]$ or as a function $[\![f]\!] : [\![X]\!] \to [\![Y]\!] + Exc$. In order to interpret the terms of Sig_{exc} one must be able to define $[\![g \circ f]\!]$ from $[\![g]\!]$ and $[\![f]\!]$ if the codomain of g is included in the domain of f, even though this is not the case for $[\![f]\!]$ and $[\![g]\!]$. This can be done thanks to the *Kleisli composition* associated using the *exception monad* $A + Exc$ [19]. Equivalently, this can be done by converting all these functions to functions from $[\![X]\!] + Exc$ to $[\![Y]\!] + Exc$:

- every function $\varphi : A \to B$ gives rise to the function

$$ {}_{\uparrow}\varphi = normal_B \circ \varphi : A \to B + Exc \tag{1}$$

- and every function $\psi : A \to B + Exc$ gives rise to the function

$$ {}^{\uparrow}\psi = [\psi | abrupt_B] : A + Exc \to B + Exc \tag{2}$$

  which is equal to ψ on A and to $abrupt_B$ on exceptions.

- It follows that every $\varphi : A \to B$ gives rise to

$$ {}^{\uparrow}({}_{\uparrow}\varphi) = [normal_B \circ \varphi | abrupt_B] = \varphi + id_{Exc} : A + Exc \to B + Exc \tag{3}$$

In this way, for each $f : X \to Y$ and $g : Y \to Z$, whatever their effects, $[\![f]\!]$ gives rise to a function from $[\![X]\!] + Exc$ to $[\![Y]\!] + Exc$ and $[\![g]\!]$ to a function from $[\![Y]\!] + Exc$ to $[\![Z]\!] + Exc$, which can always be composed.

Definition 2.3. A function $\varphi : A + Exc \to B + Exc$:

- *raises an exception* if there is some $x \in A$ such that $\varphi(x) \in Exc$.
- *recovers from an exception* if there is some $e \in Exc$ such that $\varphi(e) \in B$.
- *propagates exceptions* if it is the identity on exceptions, i.e. if $\varphi(e) = e$ for every $e \in Exc$.

Clearly, a function $\varphi : A + Exc \to B + Exc$ which propagates exceptions may raise an exception, but cannot recover from an exception. Such a function φ is characterized by its restriction $\varphi_{|_A} : A \to B + Exc$, since its restriction on exceptions $\varphi_{|_{Exc}} : Exc \to B + Exc$ is the inclusion $abrupt_B$ of Exc in $B + Exc$.

2.2 Tagging and raising exceptions: *throw*

Raising exceptions relies on the interpretation of the tagging operations.

Definition 2.4. The interpretation of the empty type $\mathbb{0}$ is the empty set $\emptyset$; thus, for each type X the interpretation of $\mathbb{0} + X$ can be identified to $[\![X]\!]$. For each exceptional type T, the interpretation of the tagging operation $tag_T : T \to \mathbb{0}$ is the coprojection function $[\![tag_T]\!] : [\![T]\!] \to Exc$, called the *tagging function* of type T.

Thus, the tagging function $[\![tag_T]\!] : [\![T]\!] \to Exc$ maps a non-exceptional value (or *parameter*) $a \in [\![T]\!]$ to an exception $[\![tag_T]\!](a) \in Exc$. This means that the non-exceptional value a in $[\![T]\!]$ gets tagged as an exception $[\![tag_T]\!](a)$ in Exc.

Now we are ready to define the raising of exceptions in a programming language.

Definition 2.5. For each exceptional type T and each type Y, the interpretation of the raising operation $throw_{T,Y}$ is the tagging function $[\![tag_T]\!]$ followed by the inclusion of Exc in $[\![Y]\!] + Exc$:

$$[\![throw_{T,Y}]\!] = abrupt_{[\![Y]\!]} \circ [\![tag_T]\!] : [\![T]\!] \to [\![Y]\!] + Exc \ .$$

2.3 Untagging and handling exceptions: *try-catch*

Handling exceptions relies on the interpretation of the untagging operations for clearing the exception tags.

Definition 2.6. For each exceptional type T, the interpretation of the untagging operation $untag_T : \mathbb{0} \to T$ is the function $[\![untag_T]\!] : Exc \to [\![T]\!] + Exc$, which satisfies for each exceptional type R:

$$\begin{cases} [\![untag_T]\!] \circ [\![tag_R]\!] = normal_{[\![T]\!]} \circ [\![cast_{R,T}]\!] : [\![R]\!] \to [\![T]\!] + Exc & \text{whenever } R \rightarrowtail T, \\ [\![untag_T]\!] \circ [\![tag_R]\!] = abrupt_{[\![T]\!]} \circ [\![tag_R]\!] = [\![throw_{R,T}]\!] : [\![R]\!] \to [\![T]\!] + Exc & \text{otherwise.} \end{cases} \tag{4}$$

It is called the *untagging function* of type T.

Thus, for each exception $e \in Exc$ the untagging function $[\![untag_T]\!](e)$ tests whether e is in $[\![R]\!]$ for some subtype R of T; if this is the case, then it returns the parameter $a \in [\![R]\!]$ such that $e = [\![tag_T]\!](a)$, otherwise it propagates the exception e. Since the domain of $[\![untag_T]\!]$ is Exc, $[\![untag_T]\!]$ is uniquely determined by its restrictions to all the exceptional types, and therefore by the above equalities.

For handling exceptions of types $T_1, \dots T_n$, raised by the interpretation of some term $f : X \to Y$ of Sig_{exc}, one provides for each k in $\{1, \dots, n\}$ a term $g_k : T_k \to Y$ of Sig_{exc} (thus, the interpretation of g_k may itself raise exceptions). Then the handling process builds a function which encapsulates some untagging functions and which propagates exceptions.

Definition 2.7. For each term $f : X \to Y$ of Sig_{exc}, each non-empty list $(g_1, \dots, g_n)$ of terms $g_k : T_k \to Y$ of Sig_{exc} where T_k is an exceptional type (for $k \in \{1, \dots, n\}$), the interpretation of the handling operation $try\{f\}\ catch\ \{T_1 \Rightarrow g_1 | \dots | T_n \Rightarrow g_n\} : X \to Y$ is the function

$$[\![try\{f\}\ catch\ \{T_1 \Rightarrow g_1 | \dots | T_n \Rightarrow g_n\}]\!] : [\![X]\!] \to [\![Y]\!] + Exc$$

defined as follows, from the interpretations of f and of the g_k's. Let $h = [\![try\{f\}\ catch\ \{T_1 \Rightarrow g_1 | \dots | T_n \Rightarrow g_n\}]\!]$, for short. For each $x \in [\![X]\!]$, $h(x) \in [\![Y]\!] + Exc$ is defined by Algorithm 1.

This definition matches that of Java exceptions as, e.g., found in [12, Ch. 14] or [2, § 5]. With the simplification of considering that inheritance and polymorphism are described by the hierarchy of exceptional types, this definition also matches that of C++ exceptions (see [1, §15] or the high-level overview of [23, §2]), or of Python [24, §7.4].

Alternatively, the interpretation $h : [\![X]\!] \to [\![Y]\!] + Exc$ of the handling operation $try\{f\}\ catch\ \{T_1 \Rightarrow g_1 | \dots | T_n \Rightarrow g_n\} : X \to Y$ can be defined in two steps, as follows (the notation is simplified by dropping the $[\![\dots]\!]$).

(try) the function $try\{f\}\, k : X \to Y + Exc$ is defined for any function $k : Exc \to Y + Exc$ by:

$$try\{f\}\, k \;=\; \Big[\, normal_Y \mid k \,\Big] \circ f$$

Algorithm 1 Interpretation of $try\{f\}\ catch\,\{T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n\}$

Require: $[\![f]\!] : [\![X]\!] \to [\![Y]\!] + Exc$; $([\![g_k]\!] : [\![T_k]\!] \to [\![Y]\!] + Exc)_{k=1,\ldots,n}$; $x \in [\![X]\!]$.
Ensure: $[\![try\{f\}\ catch\,\{T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n\}]\!](x) \in [\![Y]\!] + Exc$, denoted $h(x)$ for short.

1: let $y = [\![f]\!](x) \in [\![Y]\!] + Exc$; {First $[\![f]\!](x)$ is computed}
2: **if** $y \in [\![Y]\!]$ **then**
3: **return** $h(x) = y \in [\![Y]\!] \subseteq [\![Y]\!] + Exc$; {If y is not an exception, then it is the required result}
4: **else** {Now y is an exception}
5: **for** $k = 1, \ldots, n$ **do**
6: $z = [\![untag_{T_k}]\!](y) \in [\![T_k]\!] + Exc$; {Check whether type of y is a subtype of T_k}
7: **if** $z \in T_k$ **then return** $h(x) = [\![g_k]\!](z) \in [\![Y]\!] + Exc$; **end if** {If typeof$(y) \rightarrowtail T_k$, then y is caught}
8: **end for**
9: **return** $h(x) = y \in Exc \subseteq [\![Y]\!] + Exc$. {If the type of y is no subtype of any T_k, then y is propagated}
10: **end if**

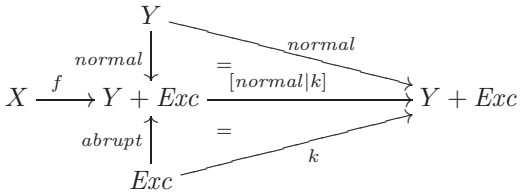
(catch) the function $catch\,\{T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n\} : Exc \to Y + Exc$ is obtained by setting $p = 1$ in the family of functions $k_p = catch\,\{T_p \Rightarrow g_p | \ldots | T_n \Rightarrow g_n\} : Exc \to Y + Exc$ (for $p = 1, \ldots, n+1$) which are defined recursively by:

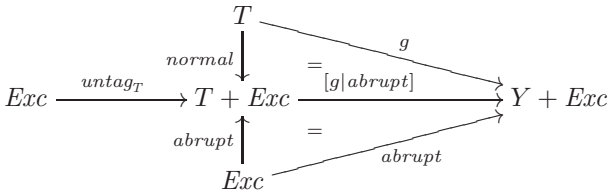
$$k_p \;=\; \begin{cases} abrupt_Y & \text{when } p = n+1 \\ [\,g_p \mid k_{p+1}\,] \circ untag_{T_p} & \text{when } p \leq n \end{cases}$$

When $n = 1$ we simply get:

$$try\{f\}\ catch\,\{T \Rightarrow g\} = \Big[normal_Y \big| [g|abrupt_Y]\Big] \circ untag_T \Big] \circ f$$

which can be illustrated as follows, with $try\{f\}\,k$ on the left and $k = catch\,\{T \Rightarrow g\}$ on the right (the subscripts are dropped):

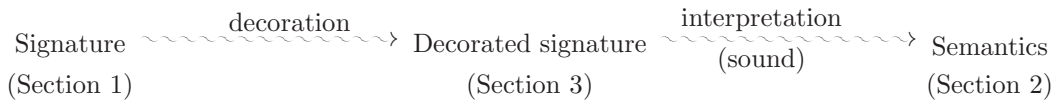




It should be noted that, in the interpretation of $try\{f\}\ catch\,\{T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n\}$, each function $[\![g_i]\!]$ may itself raise exceptions. It should also be noted that the types $T_1, \ldots, T_n$ in $try\{f\}\ catch\,\{T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n\}$ 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.

3 Decorations: from syntax to semantics

In Sections 1 and 2 we have formalized a signature for exceptions Sig_{exc} and we have described its denotational semantics. However the soundness property is not satisfied, in the sense that the denotational semantics is not a model of the signature: indeed, a term $f : X \to Y$ is not always interpreted as a function $[\![f]\!] : [\![X]\!] \to [\![Y]\!]$; it may be interpreted as $[\![f]\!] : [\![X]\!] \to [\![Y]\!] + Exc$, or even (typically when f is some untagging operation) as $[\![f]\!] : [\![X]\!] + Exc \to [\![Y]\!] + Exc$. In order to get soundness, in this Section we add *decorations* to the signature for exceptions by classifying the operations and axioms according to the interaction of their interpretations with the mechanism of exceptions.



3.1 Decorations for exceptions

By looking at the interpretation (in Section 2) of the syntax for exceptions (from Section 1), we may classify the operations and terms in three parts, depending on their interaction with the exceptions mechanism. The terms are decorated by (0), (1) and (2) used as superscripts, they are called respectively *pure* terms, *propagators* and *catchers*, according to their interpretations:

(0) the interpretation of a *pure* term may neither raise exceptions nor recover from exceptions,

(1) the interpretation of a *propagator* may raise exceptions but is not allowed to recover from exceptions,

(2) the interpretation of a *catcher* may raise exceptions and recover from exceptions.

For instance, the decoration (0) corresponds to the decoration `noexcept` in C++ (replacement of the deprecated `throw()`) and the decoration (1) corresponds to `throw(...)`, still in C++. Now the decoration (2) is usually *not* encountered in the language, since catching is usually the prerogative of the *core* untagging function, which is private, see Definition 1.3.

Similarly, we may introduce two kinds of equations between terms. This is done by using two distinct relational symbols: $\equiv$ for *strong* equations and $\sim$ for *weak* equations, which correspond to two distinct interpretations:

($\equiv$) a *strong* equation is an equality of functions both on ordinary values and on exceptions

($\sim$) a *weak* equation is an equality of functions only on ordinary values, but maybe *not on exceptions*.

Syntax		Decorated syntax		Interpretation
type	Z	type	Z	$[\![Z]\!]$
term	$X \xrightarrow{f} Y$	pure term	$X \xrightarrow{f^{(0)}} Y$	$[\![X]\!] \xrightarrow{[\![f]\!]} [\![Y]\!]$
term	$X \xrightarrow{f} Y$	propagator	$X \xrightarrow{f^{(1)}} Y$	$[\![X]\!] \xrightarrow{[\![f]\!]} [\![Y]\!] + Exc$
term	$X \xrightarrow{f} Y$	catcher	$X \xrightarrow{f^{(2)}} Y$	$[\![X]\!] + Exc \xrightarrow{[\![f]\!]} [\![Y]\!] + Exc$
equation	$f = g : X \to Y$	strong equation	$f^{(2)} \equiv g^{(2)} : X \to Y$	$[\![f]\!] = [\![g]\!]$
equation	$f = g : X \to Y$	weak equation	$f^{(2)} \sim g^{(2)} : X \to Y$	$[\![f]\!] \circ normal_{[\![X]\!]} = [\![g]\!] \circ normal_{[\![X]\!]}$

Figure 1: Interpretation of the decorated syntax.

The interpretation of these three kinds of terms and two kinds of equations is summarized in Figure 1.

This interpretation shows that any propagator can be seen as a catcher and that any pure term can be seen as a propagator and thus also as a catcher, as shown in Equations (1), (2) and (3). This allows to compose terms of different nature: $[\![h^{(2)} \circ f^{(1)}]\!] = [\![h]\!] \circ {}^{\uparrow}[\![f]\!]$, $[\![h^{(2)} \circ g^{(0)}]\!] = [\![h]\!] \circ {}^{\uparrow}[\![g]\!]$, etc. It follows that it is not a restriction to give the interpretation of the decorated equations only when both members are catchers.

3.2 Decorated proof system

The decorated proof system for exceptions is made of rules which will be used for constructing the raising and handling operations in Subsection 3.3 and for proving properties of exceptions in Appendix A. In Theorem 3.1 we prove that these rules are sound with respect to the interpretations of Figure 1.

The decorated proof system for exceptions is defined by the rules in Figure 2; the decoration properties are often grouped with other properties: for instance, "$f^{(1)} \sim g^{(1)}$" means "$f^{(1)}$ and $g^{(1)}$ and $f \sim g$"; in addition, the decoration (2) is usually dropped, since the rules assert that every term can be seen as a catcher.

Theorem 3.1. *The decorated rules for exceptions are sound with respect to the interpretation of the decorations.*

Proof. The interpretation of the decorations is defined in Figure 1. We have to check that each rule in Figure 2 is such that, whenever the interpretation of its premises is satisfied, so is the interpretation of its conclusion. This verification is now easy, some hints are given below.

(a) The first part of the decorated monadic equational rules for exceptions mean that the catchers satisfy the usual monadic equational rules with respect to the strong equations.

(a) Monadic equational rules for exceptions (first part):

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

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

(b) Monadic equational rules for exceptions (second part):

$$\frac{f^{(0)}}{f^{(1)}} \qquad \frac{f^{(1)}}{f^{(2)}} \qquad \frac{X}{id_X^{(0)}} \qquad \frac{f^{(0)} \quad g^{(0)}}{(g \circ f)^{(0)}} \qquad \frac{f^{(1)} \quad g^{(1)}}{(g \circ f)^{(1)}} \qquad \frac{f^{(1)} \sim g^{(1)}}{f \equiv g} \qquad \frac{f \equiv g}{f \sim g}$$

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

(c) Rules for the empty type $\mathbb{0}$: $\dfrac{X}{[\,]_X : \mathbb{0} \to X} \qquad \dfrac{X}{[\,]_X^{(0)}} \qquad \dfrac{f : \mathbb{0} \to Y}{f \sim [\,]_Y}$

(d) Rules for case distinction with respect to $X + \mathbb{0}$:

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

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

(e) Rules for the propagation of exceptions: $\dfrac{k^{(2)} : X \to Y}{\nabla k^{(1)} : X \to Y} \qquad \dfrac{k^{(2)} : X \to Y}{\nabla k \sim k}$

(f) Rules for the casting and tagging operations: $\dfrac{R, T \in \mathcal{T} \quad R \twoheadrightarrow T}{cast^{(0)}_{R,T} : R \to T} \qquad \dfrac{T \in \mathcal{T}}{tag^{(1)}_T : T \to \mathbb{0}}$

$$\frac{(f_T^{(1)} : T \to Y)_{T \in \mathcal{T}}}{[f_T]^{(2)}_{T \in \mathcal{T}} : \mathbb{0} \to Y} \qquad \frac{(f_T^{(1)} : T \to Y)_{T \in \mathcal{T}}}{[f_T]_{T \in \mathcal{T}} \circ tag_T \sim f_T} \qquad \frac{(f_T^{(1)} : T \to Y)_{T \in \mathcal{T}} \quad f^{(2)} : \mathbb{0} \to Y \quad \text{for all } T \in \mathcal{T}\ f \circ tag_T}{f \equiv [f_T]_{T \in \mathcal{T}}}$$

Figure 2: Decorated rules for exceptions

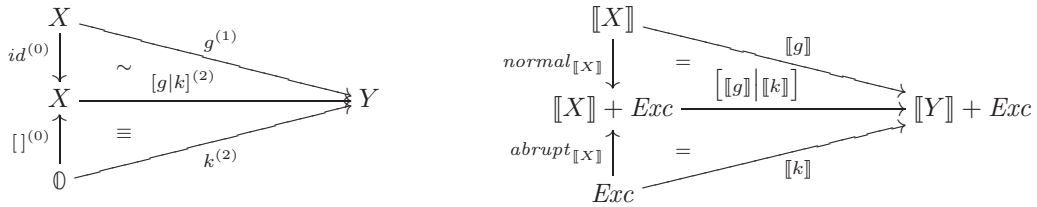
(b) The second part of the decorated monadic equational rules for exceptions deal with the conversions between decorations and with the properties of weak equations. Every strong equation is a weak one while every weak equation between propagators is a strong one. Weak equations do *not* form a congruence since the substitution rule holds only when the substituted term is pure. Indeed, let us look more closely at the interpretation of the replacement and the pure substitution rules for weak equations (note that substitution is possible only for pure terms in general since two functions

with a different behavior on exceptional values might not return the same value if an exception has occurred):

- For the replacement, let $f_1 \sim f_2 : X \to Y$. Then $[\![f_1]\!] \circ normal_{[\![X]\!]} = [\![f_2]\!] \circ normal_{[\![X]\!]} \to [\![Y]\!] + Exc$ so that for any catcher $g : Y \to Z$, interpreted as $[\![g]\!] : [\![Y]\!] + Exc \to [\![Z]\!] + Exc$, we also have $[\![g]\!] \circ [\![f_1]\!] \circ normal_{[\![X]\!]} = [\![g]\!] \circ [\![f_2]\!] \circ normal_{[\![X]\!]} : [\![X]\!] \to [\![Z]\!] + Exc$, which is the interpretation of $g \circ f_1 \sim g \circ f_2$.
- for the pure substitution, let $g_1 \sim g_2 : Y \to Z$. Then $[\![g_1]\!] \circ normal_{[\![Y]\!]} = [\![g_2]\!] \circ normal_{[\![Y]\!]} : [\![Y]\!] \to [\![Z]\!] + Exc$. Thus, for any *pure* term $f^{(0)} : X \to Y$, interpreted as $[\![f]\!] : [\![X]\!] \to [\![Y]\!]$, we have $[\![g_i^{(2)} \circ f^{(0)}]\!] = [\![g_i]\!] \circ \Uparrow [\![f]\!] = [\![g_i]\!] \circ {}^{\uparrow}({}_{\uparrow}[\![f]\!])$, so that by Equations (2) and (1), $[\![g_i^{(2)} \circ f^{(0)}]\!] \circ normal_{[\![X]\!]} = [\![g_i]\!] \circ_{\uparrow} [\![f]\!] = [\![g_i]\!] \circ normal_{[\![Y]\!]} \circ [\![f]\!]$. From $g_1 \sim g_2$, we thus have $[\![g_1]\!] \circ normal_{[\![Y]\!]} \circ [\![f]\!] = [\![g_2]\!] \circ normal_{[\![Y]\!]} \circ [\![f]\!]$, and therefore $[\![g_1^{(2)} \circ f^{(0)}]\!] \circ normal_{[\![X]\!]} = [\![g_2^{(2)} \circ f^{(0)}]\!] \circ normal_{[\![X]\!]}$ which is the interpretation of $g_1 \circ f \sim g_2 \circ f$.

(c) The interpretation of $\mathbb{0}$ and $[\,]_X^{(0)}$ are the empty set $\emptyset$ and the inclusion of $\emptyset$ in $[\![X]\!]$, respectively. Since $\mathbb{0} + Exc$ can be identified with Exc, the empty type plays an important role in the decorated proof system. For instance, a propagator $f : X \to \mathbb{0}$ is interpreted as a function $[\![f]\!] : [\![X]\!] \to Exc$ and a catcher $f : \mathbb{0} \to Y$ as a function $[\![f]\!] : Exc \to [\![Y]\!] + Exc$.

(d) For the interpretation of $[g|k]^{(2)} : X \to Y$, since $\emptyset + Exc$ can be identified with Exc we have $[\![k]\!] : Exc \to [\![Y]\!] + Exc$. Then, the interpretation of $[g|k]^{(2)}$ is the usual case distinction function $\big[\, [\![g]\!] \,\big|\, [\![k]\!] \,\big] : [\![X]\!] + Exc \to [\![Y]\!] + Exc$, i.e., the function which coincides with $[\![g]\!]$ on $[\![X]\!]$ and with $[\![k]\!]$ on Exc. This can be illustrated as follows, by a diagram in the decorated logic (on the left) and its interpretation (on the right).



(5)

(e) The rules for the propagation of exceptions build a propagator $\triangledown k^{(1)} : X \to Y$ from any catcher $k^{(2)} : X \to Y$. The interpretation of $\triangledown k^{(1)}$ is $[\![\triangledown k^{(1)}]\!] = [\![k^{(2)}]\!] \circ normal_{[\![X]\!]}$, which means that $[\![\triangledown k]\!] : [\![X]\!] \to [\![Y]\!] + Exc$ is the restriction of $[\![k]\!] : [\![X]\!] + Exc \to [\![Y]\!] + Exc$ to $[\![X]\!]$. If required, as seen in Subsection 2.1, the function $[\![\triangledown k]\!]$ is extended to $[\![X]\!] + Exc$ by propagating the exceptions; for simplicity, this function can still be denoted $[\![\triangledown k]\!]$. Thus, $[\![k]\!]$ and $[\![\triangledown k]\!]$ coincide on $[\![X]\!]$ but they differ on

Exc: if they are applied to an argument which is an exception, then $[\![k]\!]$ might catch this exception while $[\![\nabla k]\!]$ will always propagate it.

(f) These rules mean that the interpretation of the family of propagators $(tag_T : T \to \mathbb{0})_{T\in\mathcal{T}}$, which is the family $([\![tag_T]\!] : [\![T]\!] \to Exc)_{T\in\mathcal{T}}$, is a disjoint union: $Exc = \sum_{T\in\mathcal{T}} [\![T]\!]$ with the inclusions $[\![tag_T]\!]$. This allows to build a catcher with source $\mathbb{0}$ from a family of propagators with non-$\mathbb{0}$ sources: given a propagator $f_T : T \to Y$ for each exceptional type T, we get a catcher $f = [f_T]_T : \mathbb{0} \to Y$ such that $f \circ tag_T \sim f_T$ for each T, and this f is unique up to strong equations. This is interpreted as follows: given a function $[\![f_T]\!] : [\![T]\!] \to [\![Y]\!] + Exc$ for each exceptional type T, we get a unique function $[\![f]\!] : Exc \to Y + Exc$ such that $[\![f]\!] \circ [\![tag_T]\!] = [\![f_T]\!]$ for each T: indeed, since Exc is the disjoint union of the sets $[\![T]\!]$, the function $[\![f]\!]$ is defined componentwise as $[\![f_T]\!]$ on each $[\![T]\!]$. □

□

Note: the unicity rules in (c), (d) and (f) can be replaced by the following "symmetric" rules:

$$(c')\frac{f_1, f_2 : \mathbb{0} \to Y}{f_1 \sim f_2} \qquad (d')\frac{f_1, f_2 : X \to Y \quad f_1 \sim f_2 \quad f_1 \circ [\,]_X \equiv f_2 \circ [\,]_X}{f_1 \equiv f_2}$$

$$(f')\frac{f_1, f_2 : \mathbb{0} \to Y \quad \text{for all } T \in \mathcal{T}\ f_1 \circ tag_T \sim f_2 \circ tag_T}{f_1 \equiv f_2}$$

The untagging operations can be defined by applying these rules.

Proposition 3.2. *For each exceptional type T there is a catcher, denoted $untag_T^{(2)} : \mathbb{0} \to T$, unique up to strong equations, which satisfies the following weak equations:*

$$\begin{cases} untag_T^{(2)} \circ tag_R^{(1)} \sim cast_{R,T}^{(0)} : R \to T & \text{whenever } R \twoheadrightarrow T \\ untag_T^{(2)} \circ tag_R^{(1)} \sim [\,]_T^{(0)} \circ tag_R^{(1)} : R \to T & \text{otherwise} \end{cases} \tag{6}$$

Proof. Let T be some exceptional type. For each type $R \in \mathcal{T}$ we define $f_R : R \to T$ by $f_R = cast_{R,T}$ if $R \twoheadrightarrow T$ and $f_R = [\,]_T \circ tag_R$ otherwise. According to the Rules (f), each f_R is a propagator and there is a catcher $[f_R]_{R\in\mathcal{T}} : \mathbb{0} \to T$, unique up to strong equations, such that $[f_R]_{R\in\mathcal{T}} \circ tag_R \sim f_R$ for each $R \in \mathcal{T}$. Let $untag_T = [f_R]_{R\in\mathcal{T}}$. □

3.3 Decorated signature for exceptions

Now we can add decorations to the signature for exceptions.

Definition 3.3. Let $Sig_{base,\twoheadrightarrow}$ be a signature with a hierarchy of exceptional types and let Sig_{exc} be the corresponding signature for exceptions, as in Definition 1.2. The *decorated signature for exceptions* Sig_{exc}^{deco} is made of Sig_{exc} decorated as follows: the basic operations are pure and the raising and handling operations are propagators.

This decorated proof system is used now for constructing the raising and handling operations from the core tagging and untagging operations.

Definition 3.4. For each exceptional type T and each type Y, the *raising* propagator $\mathit{throw}^{(1)}_{T,Y} : T \to Y$ is defined as:

$$\mathit{throw}^{(1)}_{T,Y} = [\,]^{(0)}_Y \circ \mathit{tag}^{(1)}_T \ .$$

Definition 3.5. For each propagator $f^{(1)} : X \to Y$, each non-empty list of types $(T_1, \ldots, T_n)$ and each propagators $g^{(1)}_j : T_j \to Y$ for $j = 1, \ldots, n$, the *handling* propagator $(\mathit{try}\{f\}\ \mathit{catch}\,\{T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n\})^{(1)} : X \to Y$ is defined as:

$$\mathit{try}\{f\}\ \mathit{catch}\,\{T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n\} = \triangledown\, \mathit{TRY}\{f\}\ \mathit{catch}\,\{T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n\}$$

from a catcher $\mathit{TRY}\{f\}\ \mathit{catch}\,\{T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n\} : X \to Y$ which is defined as follows in two steps:

(try) the catcher $\mathit{TRY}\{f\}\, k : X \to Y$ is defined for any catcher $k : \mathbb{0} \to Y$ by:

$$(\mathit{TRY}\{f\}\, k)^{(2)} \ = \ \Big[\ id^{(0)}_Y \mid k^{(2)}\ \Big]^{(2)} \circ f^{(1)}$$

(catch) the catcher $\mathit{catch}\,\{T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n\} : \mathbb{0} \to Y$ is obtained by setting $p = 1$ in the family of catchers $k_p = \mathit{catch}\,\{T_p \Rightarrow g_p | \ldots | T_n \Rightarrow g_n\} : \mathbb{0} \to Y$ (for $p = 1, \ldots, n+1$) which are defined recursively by:

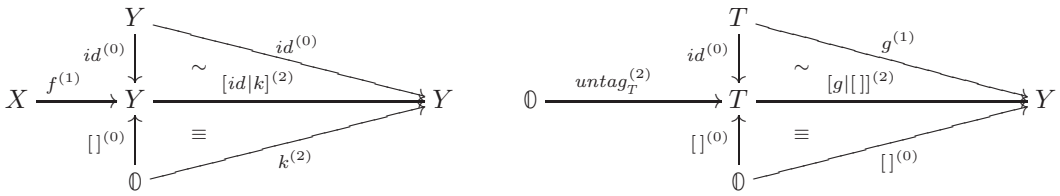
$$k^{(2)}_p \ = \ \begin{cases} [\,]^{(0)}_Y & \text{when } p = n+1 \\ \Big[\ g^{(1)}_p \mid k^{(2)}_{p+1}\ \Big]^{(2)} \circ \mathit{untag}_{T_p}(2) & \text{when } p \leq n \end{cases}$$

Since $k_{n+1} = [\,]_Y$, using the decorated rules it is easy to prove that $[g_n | k_{n+1}] \equiv g_n$ (a proof is given in appendix, Lemma A.1). It follows that when $n = 1$ and 2 we get respectively:

$$\mathit{try}\{f\}\ \mathit{catch}\,\{T \Rightarrow g\} \ \equiv\ \triangledown\,\Big(\ \big[id_Y \mid g \circ \mathit{untag}_T\big] \circ f\ \Big) \tag{7}$$

$$\mathit{try}\{f\}\ \mathit{catch}\,\{T \Rightarrow g \mid S \Rightarrow h\} \ \equiv\ \triangledown\,\Big(\ \big[id \mid [g \mid h \circ \mathit{untag}_S] \circ \mathit{untag}_T\big] \circ f\ \Big) \tag{8}$$

When $n = 1$ this can be illustrated as follows, with $\mathit{TRY}\{f\}\, k$ on the left and $k = \mathit{catch}\,\{T \Rightarrow g\}$ on the right:



Theorem 3.6. *The raising and handling constructions in Definitions 3.4 and 3.5 are sound with respect to the semantics of exceptions in Section 2.*

Proof. The proof is quite straightforward. A subtle point is that the handling mechanism is formalized by $h = try\{f\}\ catch\,\{T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n\}$ rather than $H = TRY\{f\}\ catch\,\{T_1 \Rightarrow g_1 | \ldots | T_n \Rightarrow g_n\}$. Indeed, since H is a catcher and h is the propagator defined as $h = \triangledown H$, the functions $[\![H]\!]$ and $[\![h]\!]$ coincide on $[\![X]\!]$ but not necessarily on Exc: if they are applied to an argument which is an exception, then $[\![H]\!]$ might catch this exception while $[\![h]\!]$ will always propagate it. Thus, the semantics of the handling of exceptions coincide with $[\![h]\!]$, not with $[\![H]\!]$. □

3.4 Catch all

The *catch* construction is easily extended to a *catch-all* construction, like `catch(...)` in C++, or `(except, else)` in Python. We add to the decorated logic for exceptions a pure unit type $\mathbb{1}$, which means, a type $\mathbb{1}$ such that for each type X there is a pure term $(\,)_X : X \to \mathbb{1}$, unique up to strong equations. Then we add a catcher $untag^{(2)}_{all} : \mathbb{0} \to \mathbb{1}$ with the equations $untag_{all} \circ tag_T \sim (\,)_T$ for every exceptional type T, which means that $untag_{all}$ catches exceptions of the form $tag_T(a)$ for every T and forgets the value a. For each propagators $f^{(1)} : X \to Y$ and $g^{(1)} : \mathbb{1} \to Y$, the propagator "*handle the exception e raised in f, if any, with g*" is defined as:

$$(try\{f\}\ catch\,\{all \Rightarrow g\})^{(1)} = \triangledown([id_Y \,|\, g \circ untag_{all}] \circ f) : X \to Y$$

The interpretation of the *catch-all* construction is easily obtained from this definition and from Figure 1. Since the interpretation of g is a constant, it can be identified to an element $[\![g]\!] \in [\![Y]\!] + Exc$. The function $[\![try\{f\}\ catch\,\{all \Rightarrow g\}]\!] : [\![X]\!] \to [\![Y]\!] + Exc$ is defined by Algorithm 2.

Algorithm 2 Interpretation of $try\{f\}\ catch\,\{all \Rightarrow g\}$

Require: $x \in [\![X]\!] + Exc$, $[\![f]\!] : [\![X]\!] \to [\![Y]\!] + Exc$, $[\![g]\!] \in [\![Y]\!] + Exc$.
Ensure: $[\![try\{f\}\ catch\,\{all \Rightarrow g\}]\!](x) \in [\![Y]\!] + Exc$.

1: **if** $x \in Exc$ **then return** $x \in Exc \subseteq [\![Y]\!] + Exc$; **end if** {If x is an exception, propagate it ($\triangledown$ does this)}
2: Compute $y := [\![f]\!](x) \in [\![Y]\!] + Exc$; {now x is not an exception}
3: **if** $y \in Y$ **then return** $y \in [\![Y]\!] \subseteq [\![Y]\!] + Exc$; **end if** {If $[\![f]\!](x)$ is not an exception, return normally (via id_Y)}
4: **return** $[\![g]\!] \in [\![Y]\!] + Exc$. {now $[\![f]\!](x)$ is any exception, untag (all) and return $[\![g]\!]$}

This is indeed the required semantics of the "catch-all" construction [1, §15.3.5]. It may be combined with other catchers, and it follows from this construction that every catcher following a "catch-all" is syntactically allowed, but never executed.

3.5 Higher-order constructions

The handling of exceptions can easily be extended to a functional language. In order to add higher-order features to our interpretation, let us introduce a functional type Z^W for each types W and Z. Then each $\psi : W \to Z + Exc$ gives rise to $\lambda x.\psi : \mathbb{1} \to (Z + Exc)^W$, which does not raise exceptions. It follows that the interpretation of $try\{\lambda x.\psi\}\ catch\ \{T_1 \Rightarrow g_1 | \dots | T_n \Rightarrow g_n\}$ is the same as the interpretation of $\lambda x.\psi$, which is the intended meaning of exceptions in functional languages like ML [15].

This holds for the decorated logic as well. Let us introduce a functional type $Z^{W(d)}$ for each types W and Z and each decoration (d) for terms. The interpretation of $Z^{W(0)}$ is $[\![Z]\!]^{[\![W]\!]}$, the interpretation of $Z^{W(1)}$ is $([\![Z]\!] + Exc)^{[\![W]\!]}$ and the interpretation of $Z^{W(2)}$ is $([\![Z]\!] + Exc)^{([\![W]\!]+Exc)}$. Then each $\psi^{(d)} : W \to Z$ gives rise to $\lambda x.\psi : \mathbb{1} \to Z^{W(d)}$, and a major point is that $\lambda x.\psi$ is pure for every decoration (d) of ψ. Informally, we can say that the abstraction moves the decoration from the term to the type. This means that the interpretation of $(\lambda x.\psi)^{(0)}$ depends on the decoration of ψ: for instance when $\psi^{(1)}$ is a propagator the interpretation of $(\lambda x.\psi)^{(0)}$ is $\lambda x.\psi : \mathbb{1} \to ([\![Z]\!] + Exc)^{[\![W]\!]}$. Besides, it is easy to prove in the decorated logic that, whenever f is pure, we get $try\{f\}\ catch\ \{T_1 \Rightarrow g_1 | \dots | T_n \Rightarrow g_n\} \equiv f$. It follows that this occurs when f is a lambda abstraction: $try\{\lambda x.\psi\}\ catch\ \{T_1 \Rightarrow g_1 | \dots | T_n \Rightarrow g_n\} \equiv \lambda x.\psi$, as expected in functional languages.

Conclusion and future work

We have presented a new framework for formalizing the handling of exceptions: decorations extend the syntax much like compiler qualifiers or specifiers. Decorations form a bridge between the syntax and the denotational semantics by turning the syntax sound with respect to the semantics.

The salient features of our approach are:

1. Decorating the equations allows to separate properties that are true only up to effects from generic properties.
2. There is an automatic process translating the decorated terms and properties into their interpretations.
3. We give a full system taking e.g. into account hierarchies of exceptional types, higher-order constructions, etc.
4. The proofs are in two parts: the first part is generic and can be directly dualized from the proofs on global states, the second part uses specific rules on exceptions.
5. The verification of the proofs can be done in two steps: in a first step, decorations are dropped and the proof is checked syntactically; in a second step, the decorations are taken into account in order to prove properties involving computational effects.

Our plan for future work then includes the following:

1. Our restriction to unary operations symbols in the basic signature can be dropped thanks to the notion of *sequential product* from [10].

2. We plan to extend the use of a proof assistant for checking the decorated proofs on exceptions. Indeed the decorated proof system for states, as described in [8] has been implemented in Coq [1] with the formalization of [10], so that the given proofs can be automatically verified. From this implementation, it should be possible to extract and dualize the generic part where there are correspondance between states and exceptions and then extend it to handle the full system for exceptions.

3. In the same spirit, Hilbert-Post completeness has been established for the global state effect, see [9]. Therefore, the duality should yield a similar Hilbert-Post completeness *for the core part of exceptions*. Extension to the full system for exceptions is an open problem.

Acknowledgment. We are indebted to Olivier Laurent for pointing out the extension of our approach to functional languages.

References

[1] Working Draft, Standard for Programming Language C++. ISO/IEC JTC1/SC22/WG21 standard 14882:2011. www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf

[2] Egon Börger, Wolfram Schulte, Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. Mathematical Foundations of Computer Science, MFCS'98, Brno, Czech Republic, August 24-28, 1998. LNCS vol. 1450, pp. 17–35.

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

[4] César Domínguez, Dominique Duval. A parameterization process: from a functorial point of view. International Journal of Foundations of Computer Science 23, p. 225-242 (2012).

[5] Christophe Dony. Exception Handling and Object-Oriented Programming: Towards a Synthesis. OOPSLA/ECOOP'1990. ACM Press, p. 322-330 (1990).

[6] Jean-Guillaume Dumas, Dominique Duval, Burak Ekici, Damien Pous. Formal verification in Coq of program properties involving the global state effect. arXiv:XXXX.XXXX.

[1] Effect categories and COQ, `http://coqeffects.forge.imag.fr`, [6].

[7] 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).

[8] 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).

[9] Jean-Guillaume Dumas, Dominique Duval, Samuel Mimram, Jean-Claude Reynaud. Patterns for computational effects arising from a monad or a comonad. arXiv:XXXX.XXXX.

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

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

[12] 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.

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

[14] John M. Lucassen, David K. Gifford. Polymorphic effect systems. POPL 1988. ACM Press, p. 47-57.

[15] Robin Milner, Mads Tofte, Robert Harper, David MacQueen. The Definition of Standard ML, Revised Edition. The MIT Press (1997).

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

[17] 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).

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

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

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

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

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

[23] Prakash Prabhu, Naoto Maeda and Gogul Balakrishnan. Interprocedural exception analysis for C++. ECOOP'11. Springer-Verlag Lecture Notes in Computer Science 6813, p. 583-608 (2011).

[24] Guido van Rossum and Fred L. Drake jr. Python reference manual. Centrum voor Wiskunde en Informatica, 1995.

[25] 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).

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

[27] Philip Wadler. The Marriage of Effects and Monads. ICFP 1998. ACM Press, p. 63-74 (1998).

A Some decorated proofs

The decorated rules are now used for proving some properties of exceptions. First, let us assume that the hierarchy of exceptional types is discrete (i.e., without any proper subtype). Then, we know from [7] that the tagging and untagging operations for exceptions are dual to the lookup and update operations for states. Thus, we may reuse the decorated proofs involving states from [8] for proving properties on the core part of exceptions. When the hierarchy is not discrete, the properties and their proofs have to be generalized. In addition, whether the hierarchy is discrete or not, some additional decorated proofs are needed for deriving properties of the raising and handling operations from properties of the core operations, using Definitions 3.4 and 3.5. For instance, starting from any one of the seven equations for states in [20], we can dualize this equation and derive a property about raising and handling exceptions. We give the examples of the *annihilation catch-raise* property in Subsection A.2 and of the *commutation catch-catch* property in Subsection A.3. First, a simple proof is given in Subsection A.1.

A.1 A propagator propagates

The following lemma has been used in Section 3.3. It states that given an exception, a propagator will do nothing apart from propagating it.

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

Proof. In these proofs the labels refer to the kind of rules which are used: either (a), (b), (c) or (d).
First, let us prove that $g \circ [\,]_X \equiv [\,]_Y$:

$$\dfrac{(b)\ \dfrac{(a)\ \dfrac{(c)\ \dfrac{X}{[\,]_X : 0 \to X} \qquad g : X \to Y}{g \circ [\,]_X : 0 \to Y}}{g \circ [\,]_X \sim [\,]_Y} \qquad (b)\ \dfrac{g^{(1)} \qquad \dfrac{(c)\ \dfrac{X}{[\,]_X^{(0)}}}{[\,]_X^{(1)}}}{(g \circ [\,]_X)^{(1)}} \qquad \dfrac{(c)\ \dfrac{Y}{[\,]_Y^{(0)}}}{[\,]_Y^{(1)}}}{g \circ [\,]_X \equiv [\,]_Y}$$

This first result is the unique non-obvious part in the proof of $g \equiv [g \,|\, [\,]_Y]$:

$$(d)\ \dfrac{g^{(1)} : X \to Y \qquad \dfrac{(b)\ \dfrac{(c)\ \dfrac{Y}{[\,]_Y^{(0)} : 0 \to Y}}{[\,]_Y^{(1)} : 0 \to Y}}{[\,]_Y^{(2)} : 0 \to Y} \qquad (b)\ \dfrac{g^{(1)} : X \to Y}{g^{(2)} : X \to Y} \qquad (b)\ \dfrac{g}{g \sim g} \qquad \dfrac{\vdots}{g \circ [\,]_X \equiv [\,]_Y}}{g \equiv [g \,|\, [\,]_Y]}$$

□

A.2 Annihilation handle-raise

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 [8, Proposition 3.1]. By duality we get the following *annihilation untag-tag* property (Lemma A.2), which means that tagging just after untagging, both with respect to the same exception type, returns the given exception.

Well this is true as long as there are no subtypes in exception types. Otherwise catching an exception of type $R \rightarrowtriangle T$ with $untag_T$ and then re-throwing with tag_T would slice the object of type R to become an object of the base type T, see example A.4 below.

Now, on a type without proper subtype, the result is preserved and can be used in Proposition A.3 for proving the *annihilation catch-raise* property: catching an exception and re-raising it is like doing nothing.

Lemma A.2 (Annihilation untag-tag)**.** *For each type $L \in \mathcal{T}$ without any proper subtype:*

$$tag_L^{(1)} \circ untag_L^{(2)} \equiv id_{\mathbb{0}}^{(0)} \ .$$

Now we can prove the *annihilation catch-raise* property, adding the parts about handling exceptions:

Proposition A.3 (Annihilation catch-raise)**.** *For each propagator $f^{(1)} : X \to Y$ and each type $L \in \mathcal{T}$ without any proper subtype:*

$$try\{f\}\, catch\, \{L \Rightarrow throw_{L,Y}\} \equiv f \ .$$

Proof. By Equation (7) and Definition 3.3 we have $try\{f\}\, catch\, \{L \Rightarrow throw_{L,Y}\} \equiv \triangledown([id_Y|\,]_Y \circ tag_L \circ untag_L] \circ f)$. By Lemma A.2 $[id_Y|\,]_Y \circ tag_L \circ untag_L] \equiv [id_Y|\,]_Y]$, and the Rule (d) implies that $[id_Y|\,]_Y] \equiv id_Y$. Thus $try\{f\}\, catch\, \{L \Rightarrow throw_{L,Y}\} \equiv \triangledown f$. In addition, since $\triangledown f \sim f$ and f is a propagator we get $\triangledown f \equiv f$. Finally, the transitivity of $\equiv$ yields the proposition. □

Example A.4. The latter lemma and proposition are not true anymore for general exceptional types. Indeed in this case, whenever $R \rightarrowtriangle T$, then $tag_T \circ untag_T \circ tag_R \sim tag_T \circ cast_{R,T}$. In other words, if f throws an exception of type R and $R \rightarrowtriangle T$, then $try\{f\}\, catch\, \{T \Rightarrow throw_{T,Y}\}$ would instead throw an exception of type T.

A.3 Commutation handle-handle

On states, the *commutation update-update* property means that updating two independent locations can be done in any order. By duality we get the following *commutation untag-untag* property, (Lemma A.5) which means that untagging with respect to two distinct exceptional types can be done in any order, provided that the exceptional types have no common subtype.

A detailed decorated proof of the commutation update-update property is given in [8, Proposition 3.3]. The statement of this property and its proof use *semi-pure products*, which were introduced in [10] in order to provide a decorated alternative to the strength of a monad. Dually, for the commutation untag-untag property we use *semi-pure coproducts*, thus generalizing the rules for the casting and tagging operations.

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 coproduct property with respect to 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 $f' + k' = [q_1 \circ f \mid q_2 \circ k] : A + B \to C + D$.

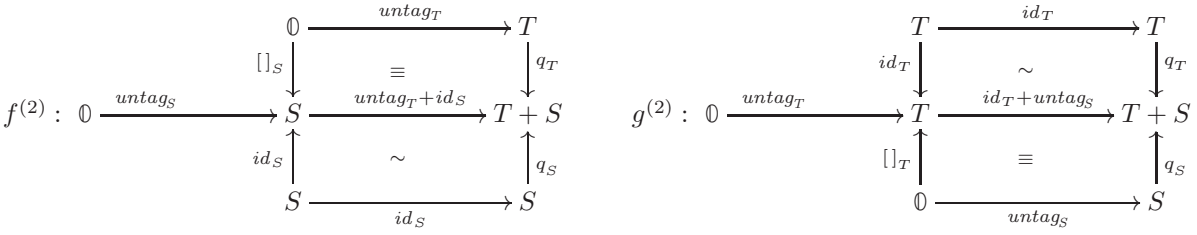
Whenever f and g are propagators it can be proved that $\triangledown [f|g] \equiv [f|g]$; thus, up to strong equations, we can assume that in this case $[f \mid g] : A + B \to C$ is a propagator; it is characterized, up to strong equations, by $[f \mid g] \circ q_1 \equiv f$ and $[f \mid g] \circ q_2 \equiv g$.

Lemma A.5 (Commutation untag-untag)**.** *For any two types T, S in $\mathcal{T}$, without any common subtype:*

$$(untag_T + id_S)^{(2)} \circ untag_S^{(2)} \equiv (id_T + untag_S)^{(2)} \circ untag_T^{(2)} : \mathbb{0} \to T + S$$

Strictly speaking, the proof of the dual lemma in [8, Proposition 3.3] is not completely applicable here since we do not suppose a discrete type hierarchy, only that the two involved types have no common subtype. Both proofs are however very close and we here provide the new one, for the sake of completeness.

Proof. Using Rule (f) of Figure 2, it is sufficient to prove that for all $R \in \mathcal{T}$, $f \circ tag_R \sim g \circ tag_R$ with f the left hand-side and g the right hand-side, as in the following diagrams:



- When $R \twoheadrightarrow T$, $(untag_T + id_S) \circ untag_S \circ tag_R \sim (untag_T + id_S) \circ [\,]_S \circ tag_R$. By the definition of the semi-pure coproduct, we have that $(untag_T + id_S) \circ [\,]_S \equiv q_T \circ untag_T$ so that $(untag_T + id_S) \circ untag_S \circ tag_R \sim q_T \circ untag_T \circ tag_R \sim q_T \circ cast_{R,T}$. We also have $(id_T + untag_S) \circ untag_T \circ tag_R \sim (id_T + untag_S) \circ cast_{R,T} \sim (id_T + untag_S) \circ id_T \circ cast_{R,T}$. The definition

of the semi-pure coproduct here yields, $(id_T + untag_S) \circ id_T \sim q_T \circ id_T$ so that, since $cast_{R,T}$ is pure, we have $(id_T + untag_S) \circ untag_T \circ tag_R \sim q_T \circ id_T \circ cast_{R,T} \sim q_T \circ cast_{R,T}$.

- The case $R \twoheadrightarrow S$ is similar.
- Finally, for $R \not\twoheadrightarrow T$ and $R \not\twoheadrightarrow S$, we have $(untag_T + id_S) \circ untag_S \circ tag_R \sim q_T \circ untag_T \circ tag_R \sim q_T \circ [\,]_T \circ tag_R \sim [\,]_{T+S} \circ tag_R$ as well as $(id_T + untag_S) \circ untag_T \circ tag_R \sim (id_T + untag_S) \circ [\,]_T \circ tag_R \sim q_S \circ untag_S \circ tag_R \sim q_S \circ [\,]_S \circ tag_R \sim [\,]_{T+S} \circ tag_R$.

□

Proposition A.6 (Commutation catch-catch)**.** *For any propagator $f^{(1)} : X \to Y$, for any two types T, S in $\mathcal{T}$ without any common subtype and any propagators $g^{(1)} : T \to Y$, $h^{(1)} : S \to Y$:*

$$try\{f\}\ catch\,\{T \Rightarrow g \mid S \Rightarrow h\} \equiv try\{f\}\ catch\,\{S \Rightarrow h \mid T \Rightarrow g\}$$

Proof. According to Equation (8): $try\{f\}\ catch\,\{T \Rightarrow g \mid S \Rightarrow h\} \equiv \triangledown([id \mid [g \mid h \circ untag_S] \circ untag_T] \circ f)$. Thus, the result will follow from $[g \mid h \circ untag_S] \circ untag_T \equiv [h \mid g \circ untag_T] \circ untag_S$. It is easy to check that $[g \mid h \circ untag_S] \equiv [g \mid h] \circ (id_T + untag_S)$, so that $[g \mid h \circ untag_S] \circ untag_T \equiv [g \mid h] \circ (id_T + untag_S) \circ untag_T$. Indeed, let $z^{(2)} : T \to Y = [g \mid h] \circ (id_T + untag_S)$. From the rules of the semi-pure coproduct, we have that $z \circ [\,] \equiv (h \circ untag_S) : \mathbb{0} \to Y$ and $z \circ id_T \sim (g \circ id_T)^{(1)} : T \to Y$. Using Rule (d) from Figure 2, this shows that $z \equiv [g \mid h \circ untag_S]$.

Similarly $[h \mid g \circ untag_T] \circ untag_S \equiv [h \mid g] \circ (untag_T + id_S) \circ untag_S$ hence $[h \mid g \circ untag_T] \circ untag_S \equiv [g \mid h] \circ (untag_T + id_S) \circ untag_S$. Then the result follows from Lemma A.5. □