Certified proofs in programs involving exceptions

Jean-Guillaume Dumas[1], Dominique Duval[1], Burak Ekici[1], and
Jean-Claude Reynaud[2]

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

Abstract. Exception handling is provided by most modern programming languages. It allows to deal with anomalous or exceptional events which require special processing. In computer algebra, exception handling is an efficient way to implement the dynamic evaluation paradigm: for instance, in linear algebra, dynamic evaluation can be used for applying programs which have been written for matrices with coefficients in a field to matrices with coefficients in a ring. Thus, a proof system for computer algebra should include a treatement of exceptions, which must rely on a careful description of a semantics of exceptions. The categorical notion of monad can be used for formalizing the raising of exceptions: this has been proposed by Moggi and implemented in Haskell. In this paper, we provide a proof system for exceptions which involves both raising and handling, by extending Moggi's approach. Moreover, the core part of this proof system is dual to a proof system for side effects in imperative languages, which relies on the categorical notion of comonad. Both proof systems are implemented in the Coq proof assistant.

1 Introduction

Using exceptions is an efficient way to simultaneously compute with dynamic evaluation in exact linear algebra. For instance, for computing the rank of a matrix, a program written for matrices with coefficients in a field can easily be modified by adding an exception treatment in order to be applied to a matrix with coefficients in a ring: the exception mechanism provides an automatic case distinction whenever it is required. The question that arises then is how to prove correctness of the new algorithm. It would be nice to design proofs with two levels, as we designed the algorithms: a first level without exceptions, then a second level taking the exceptions into account. In this paper, we propose a proof system following this principle.

Exceptions form a *computational effect*, in the sense that a syntactic expression $f : X \to Y$ is not always interpreted as a function $[\![f]\!] : [\![X]\!] \to [\![Y]\!]$ (where, as usual, the sets $[\![X]\!]$ and $[\![Y]\!]$ denote the interpretations of the types X and Y). For instance a function which raises an exception can be interpreted as a function $[\![f]\!] : [\![X]\!] \to [\![Y]\!] + \mathit{Exc}$ 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.

The fundamental computational effect is the evolution of states in an imperative language, when seen from a functional point of view. There are several frameworks for combining functional and imperative programming: the effect systems classify the expressions according to the way they interact with the store [11], while the Kleisli category of the monad of states $(X \times St)^{St}$ (where St is the set of states) also provides a classification of expressions [14]; indeed, both approaches are related [20]. Lawvere theories were proposed for dealing with the operations and equations related to computational effects [16,12]. Another related approach, based on the fact that the state is observed, relies on the classification of expressions provided by the coKleisli category of the comonad of states $X \times St$ and its associated Kleisli-on-coKleisli category [4].

The treatment of exceptions is another fundamental computational effect. It can be studied from the point of view of the monad of exceptions $X + Exc$ (where Exc is the set of exceptions), or with a Lawvere theory, however in these settings it is difficult to handle exceptions because this operation does not have the required algebraicity property [17,18], in contrast with raising exceptions, looking up and updating states. This makes exceptions much more difficult to formalize than other effects in the Lawvere theory framework. This issue has been circumvented in [19] in order to get a Hoare logic for exceptions, in [13] by using both algebras and coalgebras and in [18] by introducing handlers. The formalization of exceptions can also be made from a coalgebraic point of view [10]. In this paper we extend Moggi's original approach: we use the classification of expressions provided by the Kleisli category of the monad of exceptions and its associated coKleisli-on-Kleisli category; moreover, we use the duality between states and exceptions discovered in [3]. However, it is not necessary to know about monads or comonads for reading this paper: the definitions and results are presented in an elementary way, in terms of equational theories.

In Section 2 we give a motivating example for the use of exceptions as an efficient way to compute with dynamic evaluation in exact linear algebra. Then in Section 3 we define the syntax of a simple language for dealing with exceptions. The intended denotational semantics is described in Section 4: we dissociate the core operations for switching between exceptions and non-exceptional values, on one side, from their encapsulation in the raising and handling operations, on the other side. In Section 5 we add decorations to the 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 (e.g., like the `throw` annotation in C++ functions' signatures). In addition, we also decorate the equations, which provides a proof system for dealing with this decorated syntax. The main result of the paper is that this proof system is sound with respect to the semantics of exceptions (Theorem 1). A major property of this new proof system is that we can separate the verification of the proofs in two steps: a first step checks properties of the programs whenever there is no effect, while a second step takes the effects into account via the decorations.

Several properties of exceptions have been proven using this proof system, and these proofs have been verified in the Coq proof assistant.

2 Rank computations modulo composite numbers

Rank algorithms play a fundamental role in computer algebra. For instance, computing homology groups of simplicial complexes reduces to computing ranks and integer Smith normal forms of the associated boundary matrices [6]. One of the most efficient method for computing the Smith normal form of such boundary matrices also reduces to computing ranks but modulo the valence, a product of the primes involved in the Smith form [7]. Now rank algorithms (mostly Gaussian elimination and Krylov based methods) work well over fields (note that Gaussian elimination can be adapted modulo powers of primes [7, §5.1]). Modulo composite numbers, zero divisors arise. Gauss-Bareiss method could be used but would require to know the determinant in advance, with is more difficult than the valence. The strategy used in [7] is to factor the valence, but only partially (factoring is still not polynomial). The large remaining composite factors will have very few zero divisors and thus Gaussian elimination or Krylov methods will have very few risks of falling upon them. Thus one can use dynamic evaluation: try to launch the rank algorithm modulo this composite number with large prime factors and "pretend" to be working over a field [8]. In any case, if a zero divisor is encoutered, then the valence has been further factored (in polynomial time!) and the algorithm can split its computation independently modulo both factors. An effective algorithm design, here in C++, enabling this dynamic evaluation with very little code modification, uses exceptions:

1. Add one exception at the arithmetic level, for signaling a tentative division by a zero divisor, see Fig. 1.

```
inline Integer invmod(const Integer& a, const Integer& m) {
    Integer gcd,u,v; ExtendedEuclideanAlgorithm(gcd,u,v,a,m);
    if (gcd != 1) throw ZmzInvByZero(gcd);
    return u>0?u:u+=m;
}
```

Fig. 1. Throwing an exception upon division by a non unit

2. Catch this exception inside the rank algorithm and throw a new exception with the state of the rank iteration, see Fig. 2 (in our implementation the class `zmz` wraps integers modulo m, the latter modulus being a `static` global variable).
3. Then it is sufficient to wrap the rank algorithm with a dynamic evaluation, splitting the continuation modulo both factors, see Fig. 3.

```
try {
      invpiv = zmz(1) / A[k][k];
} catch (ZmzInvByZero e) {
      throw GaussNonInvPivot(e.getGcd(), k, currentrank);
}
```

Fig. 2. Re-throwing an exception to forward rank and iteration number information

```
void DynEvalRank(RankPairs& vectranks, size_t addedrank, ZmzMatrix& A) {
  try {
    long rank = gaussrank(A);        // in place modifications of A
    vectranks.push_back(RankPair(rank+addedrank, zmz::getModulus()) );
  } catch (GaussNonInvPivot e) {  // Split

    long mymodulus1 = zmz::getModulus()/e.getGcd();
    long mymodulus2 = zmz::getModulus()/mymodulus1;
     // Homomorphism on the (n-k)×(m-k) remaining block
    zmz::setModulus( mymodulus1 );
    ZmzMatrix M1(A,e.getIndex(),e.getIndex());
    DynEvalRank(vectranks, e.getCurrentRank(), M1);
     // Homomorphism on the (n-k)×(m-k) remaining block
    zmz::setModulus( mymodulus2 );
    ZmzMatrix M2(A,e.getIndex(),e.getIndex());
    DynEvalRank(vectranks, e.getCurrentRank(), M2);
  } }
```

Fig. 3. Recursive splitting wrapper around classical Gaussian elimination, packing a list of pairs of rank and associated modulus.

The advantage of using exceptions over other software design is twofold: first, throwing an exception at the arithmetic level and not only on the inverse call in the rank algorithm allows to prevent that other unexpected divisions by zero divisors go unnoticed; second, re-throwing a new exception in the rank algorithm allows to keep its specifications unchanged. It also enables to keep the modifications of rank algorithms to a minimum and to clearly separate normal behavior with primes from the exceptional handling of splitting composite moduli.

In the following, we propose a proof system with decorations, so that proofs can easily be made in two steps: a first step without exceptions, that is, just preserving an initial proof of the rank algorithm; then a second level taking the exceptions into account.

3 Syntax 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`, `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 language we develop is quite close to the Ynot language [15]. Like Ynot, our approach is "a generalization of the well-known ideas from type-and-effect systems and their monadic counterparts". However, we can use comonads as well as monads, and we offer more modularity: in Ynot there is a single monad encompassing all effects, while we are able to compose effects, thus enabling to prove programs in several simpler stages.

The *basic* part of our syntax is a signature Sig_{base}, made of *types* (or *sorts*) and *operations*. 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 there is a base class for exceptional types, such that the exceptional types are precisely the subtypes of this base class. Now, we assume that some basic signature Sig_{base} and some set of exceptional types $\mathcal{T}$ have been chosen.

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

Definition 1. *The* signature for exceptions *Sig_{exc} is made of Sig_{base} with the following operations: a* raising *operation for each exceptional type T and each type Y:*

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

and a handling *operation for each Sig_{exc}-term $f : X \to Y$, each non-empty list of exceptional types $(T_i)_{1\le i\le n}$ and each family of Sig_{exc}-terms $(g_i : T_i \to Y)_{1\le i\le n}$:*

$$try\{f\}\ catch\ \{T_i \Rightarrow g_i\}_{1\le i\le n} : X \to Y \ .$$

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. This means that the operations for catching exceptions are *private* operations: they are not part of the signature for exceptions. More precisely, the operations for raising and handling exceptions can be expressed in terms of a private *empty type* and two families of private operations: the *tagging* operations for creating exceptions and the *untagging* operations for catching them (inside the *catch* block of any *try-catch* expression). The tagging and untagging operations are called the *core* *operations* for exceptions. They are not part of Sig_{exc}, but the interpretations of the operations for raising and handling exceptions, which are part of Sig_{exc}, are defined in terms of the interpretations of the core operations. The meaning of the core operations is given in Section 4.

Definition 2. *Let Sig_{exc} be the signature for exceptions. The* core *of Sig_{exc} is the signature Sig_{core} made of Sig_{base} with a type $\mathbb{0}$ called the* empty type *and two operations for each exceptional type T:*

$$tag_T : T \to \mathbb{0} \quad \textit{and} \quad untag_T : \mathbb{0} \to T$$

where tag_T is called the exception constructor *or the* tagging *operation and $untag_T$ is called the* exception recovery *or the* untagging *operation.*

4 Denotational semantics for exceptions

In this Section we define a denotational semantics for exceptions which relies on the common semantics of exceptions in various languages, for instance in C++ [1, Ch. 15], Java [9, Ch. 14] or ML.

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 $[\![f]\!] : [\![X]\!] \to [\![Y]\!]$. But, when $f : X \to Y$ in Sig_{exc} is a raising or handling operation, or when $f : X \to Y$ in Sig_{core} is a tagging or untagging operation, it is not interpreted as a function $[\![f]\!] : [\![X]\!] \to [\![Y]\!]$: this corresponds to the fact that the exceptions form a *computational effect*. The distinction between ordinary and exceptional values is discussed in Subsection 4.1. Then, denotational semantics of raising and handling exceptions are considered in Subsections 4.2 and 4.3, respectively. We assume that some interpretation of Sig_{base} has been chosen.

4.1 Ordinary values and exceptional values

In order to express the denotational semantics of exceptions, a fundamental point is to distinguish 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*. Let Exc be a set, called the *set of exceptions*. For each set A, the set $A + Exc$ is the disjoint union of A and Exc and the canonical inclusions are denoted $normal_A : A \to A + Exc$ and $abrupt_A : Exc \to A + Exc$. For each functions $f : A \to B$ and $g : Exc \to B$, we denote by $[f|g] : A + Exc \to B$ the unique function such that $[f|g] \circ normal_A = f$ and $[f|g] \circ abrupt_A = g$.

Definition 3. *An element of $A + Exc$ is an* ordinary value *if it is in A and an* exceptional value *if it is in Exc. 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 $e \in \mathit{Exc}$ such that $\varphi(e) \in B$.*
- propagates exceptions *if $\varphi(e) = e$ for every $e \in \mathit{Exc}$.*

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

In the denotational semantics for exceptions, we will see that a term $f : X \to Y$ of $\mathit{Sig}_{\mathit{exc}}$ or $\mathit{Sig}_{\mathit{core}}$ may be interpreted either as a function $\llbracket f \rrbracket : \llbracket X \rrbracket \to \llbracket Y \rrbracket$ or as a function $\llbracket f \rrbracket : \llbracket X \rrbracket \to \llbracket Y \rrbracket + \mathit{Exc}$ or as a function $\llbracket f \rrbracket : \llbracket X \rrbracket + \mathit{Exc} \to \llbracket Y \rrbracket + \mathit{Exc}$. However, in all cases, it is possible to convert $\llbracket f \rrbracket$ to a function from $\llbracket X \rrbracket + \mathit{Exc}$ to $\llbracket Y \rrbracket + \mathit{Exc}$, as follows.

Definition 4. *The* upcasting conversions *are the following transformations:*

- *every function $\varphi : A \to B$ gives rise to ${}_{\uparrow}\varphi = \mathit{normal}_B \circ \varphi : A \to B + \mathit{Exc}$,*
- *every function $\psi : A \to B + \mathit{Exc}$ gives rise to ${}^{\uparrow}\psi = [\psi | \mathit{abrupt}_B] : A + \mathit{Exc} \to B + \mathit{Exc}$, which is equal to ψ on A and which propagates exceptions;*
- *it follows that every function $\varphi : A \to B$ gives rise to $\Uparrow\varphi = {}^{\uparrow}({}_{\uparrow}\varphi) = [\mathit{normal}_B \circ \varphi | \mathit{abrupt}_B] = \varphi + \mathit{id}_{\mathit{Exc}} : A + \mathit{Exc} \to B + \mathit{Exc}$, which is equal to φ on A and which propagates exceptions.*

Since the upcasting conversions are *safe* (i.e., injective), when there is no ambiguity the symbols ${}_{\uparrow}$, ${}^{\uparrow}$ and $\Uparrow$ may be omitted. In this way, for each $f : X \to Y$ and $g : Y \to Z$, whatever their effects, we get $\llbracket f \rrbracket : \llbracket X \rrbracket + \mathit{Exc} \to \llbracket Y \rrbracket + \mathit{Exc}$ and $\llbracket g \rrbracket : \llbracket Y \rrbracket + \mathit{Exc} \to \llbracket Z \rrbracket + \mathit{Exc}$, which can be composed. Thus, every term of $\mathit{Sig}_{\mathit{exc}}$ and $\mathit{Sig}_{\mathit{core}}$ can be interpreted by first converting the interpretation of each of its components $f : X \to Y$ to a function $\llbracket f \rrbracket : \llbracket X \rrbracket + \mathit{Exc} \to \llbracket Y \rrbracket + \mathit{Exc}$. For $\mathit{Sig}_{\mathit{exc}}$, this coincides with the *Kleisli composition* associated to the *exception monad* $A + \mathit{Exc}$ [14].

We will also use the following conversion.

Definition 5. *The* downcasting *conversion is the following transformation:*

- *every function $\theta : A + \mathit{Exc} \to B + \mathit{Exc}$ gives rise to ${}^{\downarrow}\theta = \theta \circ \mathit{normal}_A : A \to B + \mathit{Exc}$ which is equal to θ on A and which propagates exceptions.*

This conversion is unsafe: different θ's may give rise to the same ${}^{\downarrow}\theta$.

4.2 Tagging and raising exceptions

Raising exceptions relies on the interpretation of the tagging operations. 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 with $\llbracket X \rrbracket$.

Definition 6. *Let Exc be the disjoint union of the sets $\llbracket T \rrbracket$ for all the exceptional types T. Then, for each exceptional type T, the interpretation of the tagging operation $\mathit{tag}_T : T \to \mathbb{0}$ is the coprojection function*

$$\llbracket \mathit{tag}_T \rrbracket : \llbracket T \rrbracket \to \mathit{Exc}\ .$$

Thus, the tagging function $[\![tag_T]\!] : [\![T]\!] \to Exc$ maps a non-exceptional value (or *parameter*) $a \in [\![T]\!]$ to an exception $[\![tag_T]\!](a) \in Exc$. We can now define the raising of exceptions in a programming language.

Definition 7. *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 $[\![Y]\!] + Exc$:*

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

4.3 Untagging and handling exceptions

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

Definition 8. *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:

$$[\![untag_T]\!] \circ [\![tag_R]\!] = \begin{cases} normal_{[\![T]\!]} & \text{when } R = T \\ abrupt_{[\![T]\!]} \circ [\![tag_R]\!] & \text{when } R \neq T \end{cases} : [\![R]\!] \to [\![T]\!] + Exc.$$

Thus, the untagging function $[\![untag_T]\!]$, when applied to any exception e, first tests whether e is in $[\![T]\!]$; if this is the case, then it returns the parameter $a \in [\![T]\!]$ 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 equalities in Definition 8. 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 i in $\{1, \dots, n\}$ a term $g_i : T_i \to Y$ of Sig_{exc} (thus, the interpretation of g_i may itself raise exceptions). Then the handling process builds a function which first executes f, and if f returns an exception then maybe catches this exception. The catching part encapsulates some untagging functions, but the resulting function always propagates exceptions.

Definition 9. *For each term $f : X \to Y$ of Sig_{exc}, and each non-empty lists $(T_i)_{1 \leq i \leq n}$ of exceptional types and $(g_i : T_i \to Y)_{1 \leq i \leq n}$ of terms of Sig_{exc}, let $(recover_i)_{1 \leq i \leq n}$ denote the family of functions defined recursively by:*

$$recover_i\ =\ \begin{cases} [\ [\![g_n]\!] \mid abrupt_Y\ ] \circ untag_{T_n} & \text{when } i = n \\ [\ [\![g_i]\!] \mid recover_{i+1}\ ] \circ untag_{T_i} & \text{when } i < n \end{cases} : Exc \to Y + Exc$$

Then the interpretation of the handling operation is:

$$[\![try\{f\}\ catch\ \{T_i \Rightarrow g_i\}_{1 \leq i \leq n}]\!] = [\ normal_Y \mid recover_1\ ] \circ [\![f]\!].$$

It should be noted that $[\![f]\!] : [\![X]\!] \to [\![Y]\!] + \mathit{Exc}$ and that similarly $[\![\mathit{try}\{f\}\ \mathit{catch}\ \{T_i \Rightarrow g_i\}_{1\le i\le n}]\!] : [\![X]\!] \to [\![Y]\!] + \mathit{Exc}$. When $n = 1$ we get:

$$[\![\mathit{try}\{f\}\ \mathit{catch}\ \{T \Rightarrow g\}]\!] = [\ \mathit{normal}_Y \mid [\ [\![g]\!] \mid \mathit{abrupt}_Y\ ] \circ \mathit{untag}_T\ ] \circ [\![f]\!]\ .$$

This definition matches that of Java exceptions [9, Ch. 14] or C++ exceptions [1, §15]. In particular, in the interpretation of $\mathit{try}\{f\}\ \mathit{catch}\ \{T_i \Rightarrow g_i\}_{1\le i\le n}$, each function $[\![g_i]\!]$ may itself raise exceptions; and the types $T_1, \dots, T_n$ need not be pairwise distinct, but if $T_i = T_j$ for some $i < j$ then g_j is never executed.

5 A decorated equational proof system for exceptions

In Sections 3 and 4 we have formalized the signature for exceptions Sig_{exc}, its associated core signature Sig_{core}, and we have described their denotational semantics. However the soundness property is not satisfied, in the sense that the denotational semantics is not a model of the signature, in the usual sense: indeed, a term $f : X \to Y$ is not always interpreted as a function $[\![f]\!] : [\![X]\!] \to [\![Y]\!]$; it may be interpreted as $[\![f]\!] : [\![X]\!] \to [\![Y]\!] + \mathit{Exc}$, or as $[\![f]\!] : [\![X]\!] + \mathit{Exc} \to [\![Y]\!] + \mathit{Exc}$. In order to get soundness, in this Section we add *decorations* to the signature for exceptions by classifying the operations and equations according to the interaction of their interpretations with the mechanism of exceptions.

$$\begin{tikzcd}\begin{array}{c}\text{Signature}\\ \text{(Section 3)}\end{array} \arrow[r, rightsquigarrow, "decoration"] & \begin{array}{c}\text{Decorated signature}\\ \text{(Section 5)}\end{array} \arrow[r, rightsquigarrow, "interpretation", "\text{(sound)}"'] & \begin{array}{c}\text{Semantics}\\ \text{(Section 4)}\end{array}\end{tikzcd}$$

5.1 Decorations for exceptions

By looking at the interpretation (in Section 4) of the syntax for exceptions (from Section 3), we get a classification of 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++. The decoration (2) is usually *not* encountered in the language, since catching is the prerogative of the *core* untagging function, which is private. Similarly, we introduce two kinds of equations between terms. This is done by using two distinct relational symbols 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*.

The interpretation of these three kinds of terms and two kinds of equations is summarized in Fig. 4. It has been shown in Section 4.1 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. This allows to compose terms of different nature, so that it is not a restriction to give the interpretation of the decorated equations only when both members are catchers.

Syntax	Decorated syntax	Interpretation
type X	type X	$[\![X]\!]$
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 $f^{(2)} \sim g^{(2)} : X \to Y$	$[\![f]\!] \circ normal_{[\![X]\!]} = [\![g]\!] \circ normal_{[\![X]\!]}$

Fig. 4. Interpretation of the decorated syntax.

Now we can add decorations to the signature for exceptions Sig_{exc} and its associated core signature Sig_{core}, from Definitions 1 and 2.

Definition 10. *The* decorated signature for exceptions Sig^{deco}_{exc} *and its associated* decorated core signature Sig^{deco}_{core} *are made of* Sig_{exc} *and* Sig_{core}*, respectively, decorated as follows: the basic operations are pure, the tagging, raising and handling operations are propagators, and the untagging operations are catchers.*

5.2 Decorated rules for exceptions

In this Section we define an equational proof system for exceptions. This proof system is made of the rules in Fig. 5. It can be used for proving properties of exceptions, for instance in the Coq proof assistant. In Fig. 5, 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; and several rules with the same premisses may be grouped together: $\frac{H_1 \dots H_n}{C_1 \dots C_p}$ stands for $\frac{H_1 \dots H_n}{C_1}, \dots, \frac{H_1 \dots H_n}{C_p}$.

Rules (a) are the usual rules for the monadic equational logic; they are valid for all decorated terms and for strong equations. Rules (b) provide more information on the decorated monadic equational logic for exceptions; in particular, the substitution of f in a weak equation $g_1 \sim g_2$ holds only when f is pure, which is quite restrictive. Rules (c) ensure that the empty type is a kind of initial type with respect to weak equations. Rules (d) are used for case distinctions between exceptional arguments (on the "$\mathbb{0}$" side of $X + \mathbb{0}$) and non-exceptional arguments (on the "X" side of $X + \mathbb{0}$). The symbol $^{\downarrow}$ in rules (e) is interpreted as the downcast conversion, see Definition 5. Rules in (f) mean that the tagging operations are interpreted as the canonical inclusions of the exceptional types in the set *Exc*, see Definition 6. The rules in (g) determine the untagging operations up to strong equations: an untagging operation recovers the exception parameter whenever the exception type is matched, and it propagates the exception otherwise see Definition 8.

Remark 1. It has been shown in [3] that the denotational semantics of the core language for exceptions is dual to the denotational semantics for states: the *tagging* and *untagging* operations are respectively dual to the *lookup* and *update* operations. In fact, this duality is also valid for the decorated equational logics.

This decorated proof system is used now (in Definition 12) for constructing the raising and handling operations from the core tagging and untagging operations. It has to be noted that the term $\mathit{catch}\,\{T_i \Rightarrow g_i\}_{1\leq i\leq n} \circ f$ may catch exceptions, while the handling operation, which coincides with $\mathit{catch}\,\{T_i \Rightarrow g_i\}_{1\leq i\leq n} \circ f$ on non-exceptional values, must propagate exceptions; this is why the downcast operator $^{\downarrow}$ is used.

Definition 11. *For each exceptional type T and each type Y, the* raising *operation $\mathit{throw}^{(1)}_{T,Y} : T \to Y$ is the propagator defined as:*

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

Definition 12. *For each propagator $f^{(1)} : X \to Y$, each non-empty lists of types $(T_i)_{1\leq i\leq n}$ and propagators $(g_i^{(1)} : T_i \to Y)_{1\leq i\leq n}$, let $\mathit{catch}\,\{T_i \Rightarrow g_i\}^{(2)}_{1\leq i\leq n} : Y \to Y$ denote the catcher defined by:*

$$\mathit{catch}\,\{T_i \Rightarrow g_i\}^{(2)}_{1\leq i\leq n} = [\ \mathit{id}_Y \mid \mathit{recover}_1\ ]$$

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

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

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

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

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

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

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

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

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

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

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

$$\frac{f, g : X \to Y \quad f \sim g \quad f \circ [\,]_X \equiv g \circ [\,]_X}{f \equiv g}$$

(e) Propagating exceptions: $\quad \frac{k^{(2)} : X \to Y}{({\downarrow}k)^{(1)} : X \to Y \quad {\downarrow}k \sim k}$

(f) Tagging:

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

$$\frac{f, g : \mathbb{0} \to Y \quad f \circ tag_T \sim g \circ tag_T \text{ for all } T \in \mathcal{T}}{f \equiv g}$$

(g) Untagging:

$$\frac{T \in \mathcal{T}}{untag_T^{(2)} : \mathbb{0} \to T \quad untag_T \circ tag_T \sim id_T} \qquad \frac{R, T \in \mathcal{T} \quad R \neq T}{untag_T \circ tag_R \sim [\,]_T \circ tag_R}$$

Fig. 5. Decorated rules for exceptions

where $(recover_i^{(2)} : \mathbb{0} \to Y)_{1\leq i\leq n}$ *denotes the family of catchers defined recursively by:*

$$recover_i^{(2)} \;=\; \begin{cases} g_n \circ untag_{T_n} & \text{when } i = n, \\ [g_i \mid recover_{i+1}\,] \circ untag_{T_i} & \text{when } i < n. \end{cases}$$

Then, the handling *operation* $(try\{f\}\ catch\ \{T_i \Rightarrow g_i\}_{1\leq i\leq n})^{(1)} : X \to Y$ *is the propagator defined by:*

$$try\{f\}\ catch\ \{T_i \Rightarrow g_i\}_{1\leq i\leq n} = {\downarrow}(catch\ \{T_i \Rightarrow g_i\}_{1\leq i\leq n} \circ f)$$

When $n = 1$ we get:

$$try\{f\}\ catch\ \{T \Rightarrow g\} = {\downarrow}(catch\ \{T \Rightarrow g\} \circ f) = {\downarrow}([\ id_Y \mid g \circ untag_T\ ] \circ f).$$

Now Theorem 1 derives easily, by induction, from Fig. 5 and 4 and from Definitions 11 and 12.

Theorem 1. *The decorated rules for exceptions and the raising and handling constructions are sound with respect to the denotational semantics of exceptions.*

5.3 A decorated proof: a propagator propagates

With these tools, it is now possible to prove properties of programs involving exceptions and to check these proofs in Coq. For instance, let us *prove* that given an exception, a propagator will do nothing apart from propagating it. Recall that the interpretation of $[\,]_Z$ (or, more precisely, of $\uparrow[\,]_Z$) is $abrupt_{[\![Z]\!]} : Exc \to [\![Z]\!] + Exc$.

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

Proof. This lemma can be proved as follows; the labels refer to Fig. 5, and their subscripts to the proof in Coq of Fig. 6.

$$(b)_4\ \dfrac{(c)_2\ \dfrac{(a)\ \dfrac{(c)_1\ \dfrac{X}{[\,]_X : \mathbb{0} \to X} \qquad g : X \to Y}{g \circ [\,]_X : \mathbb{0} \to Y}}{g \circ [\,]_X \sim [\,]_Y} \qquad (b)_2\ \dfrac{g^{(1)} \qquad (b)_1\ \dfrac{(c)_3\ \dfrac{X}{[\,]_X^{(0)}}}{[\,]_X^{(1)}}}{(g \circ [\,]_X)^{(1)}} \qquad (b)_3\ \dfrac{(c)_4\ \dfrac{Y}{[\,]_Y^{(0)}}}{[\,]_Y^{(1)}}}{g \circ [\,]_X \equiv [\,]_Y}$$

The proof in Coq follows the same line as the mathematical proof above. It goes as in Figure 6. The Coq library for exceptions, EXCEPTIONS-0.1.tar.gz, can be found online: http://coqeffects.forge.imag.fr (in file `Proofs.v`) with proofs of many other properties of programs involving exceptions.

It should be recalled that any Coq proof is read from bottom up. The application of the `from_empty_is_weakly_unique` rule certifies that the term `g o (@empty X)`, because its domain is the empty type, is weakly equal to the term `(@empty Y)`. Thus, we have the left side sub-proof: `g o (@empty X)` $\sim$ `(@empty Y)`. There, weak equality is converted into strong: applying `propagator_weakeq_is_strongeq` resolves the goal: `g o (@empty X) == (@empty Y)` and produces as sub-goals that there is no catcher involved in both hand sides (middle and right side sub-proofs).

```
Lemma propagator_propagates_1:
forall X Y (g: term Y X),
is propagator g -> g o (@empty X) == (@empty Y).
Proof.
    intros.
    (* (b)4 *) apply propagator_weakeq_is_strongeq.
    (* (b)2 *) apply is_comp. assumption.
    (* (b)1 *) apply is_pure_propagator. (* (c)3 *) apply is_empty.
    (* (b)3 *) apply is_pure_propagator. (* (c)4 *) apply is_empty.
    (* (c)2 *) apply from_empty_is_weakly_unique.
Qed.
```

Fig. 6. Proof in Coq that “a propagator propagates exceptions”

5.4 A hierarchy of exceptional types

In object-oriented languages, exceptions are usually the objects of classes which are related by a hierarchy of subclasses. Our framework can be extended in this direction by introducing a hierarchy of exceptional types: the set $\mathcal{T}$ is endowed with a partial order $\rightarrowtriangle$ called the *subtyping relation*, and the signature $\mathit{Sig}_{\mathit{base}}$ is extended with a *cast* operation $\mathit{cast}_{R,T} : R \to T$ whenever $R \rightarrowtriangle T$. The interpretation of $\mathit{cast}_{R,T}$ is a pure function $\llbracket \mathit{cast}_{R,T} \rrbracket : \llbracket R \rrbracket \to \llbracket T \rrbracket$, such that $\llbracket \mathit{cast}_{T,T} \rrbracket$ is the identity on $\llbracket T \rrbracket$ and when $S \rightarrowtriangle R \rightarrowtriangle T$ then $\llbracket \mathit{cast}_{S,T} \rrbracket = \llbracket \mathit{cast}_{R,T} \rrbracket \circ \llbracket \mathit{cast}_{S,R} \rrbracket$. Definition 8 has to be modified as follows: the function $\llbracket \mathit{untag}_T \rrbracket : \mathit{Exc} \to \llbracket T \rrbracket + \mathit{Exc}$ satisfies for each exceptional type R:

$$\llbracket \mathit{untag}_T \rrbracket \circ \llbracket \mathit{tag}_R \rrbracket = \begin{cases} \mathit{normal}_{\llbracket T \rrbracket} \circ \llbracket \mathit{cast}_{R,T} \rrbracket & \text{when } R \rightarrowtriangle T \\ \mathit{abrupt}_{\llbracket T \rrbracket} \circ \llbracket \mathit{tag}_R \rrbracket & \text{otherwise} \end{cases} : \llbracket R \rrbracket \to \llbracket T \rrbracket + \mathit{Exc}.$$

6 Conclusion

Exceptions are part of most modern programming languages and they are useful in computer algebra, typically for implementing dynamic evaluation. We have presented a new framework for formalizing the treatment of exceptions. These decorations form a bridge between the syntax and the denotational semantics by turning the syntax sound with respect to the semantics, without adding any explicit “type of exceptions” as a possible return type for the operations. The salient features of our approach are:

- We provide rules for equational proofs on programs involving exceptions (Fig. 5) and an automatic process for interpreting these proofs (Fig. 4).
- Decorating the equations allows to separate properties that are true only up to effects (weak equations) from properties that are true even when effects are considered (strong equations).

- Moreover, 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.
- The distinction between the language for exceptions and its associated private core language (Definitions 1 and 2) allows to split the proofs in two successive parts; in addition, the private part can be directly dualized from the proofs on global states (relying on [3] and [4]).
- A proof assistant can be used for checking the decorated proofs on exceptions. Indeed the decorated proof system for states, as described in [4] has been implemented in Coq [2] and dualized for exceptions (see `http://coqeffects.forge.imag.fr`).

We have used the approach of decorated logic, which provides rules for computational effects by adding decorations to usual rules for "pure" programs. This approach can be extended in order to deal with multivariate operations [5], conditionals, loops, and high-order functions.

References

1. Working Draft, Standard for Programming Language C++. ISO/IEC JTC1/SC22/WG21 standard 14882:2011.
2. Jean-Guillaume Dumas, Dominique Duval, Burak Ekici, Damien Pous. Formal verification in Coq of program properties involving the global state effect. JFLA'13, Fréjus, France, 2013. arXiv:1310.0794.
3. Jean-Guillaume Dumas, Dominique Duval, Laurent Fousse, Jean-Claude Reynaud. A duality between exceptions and states. Mathematical Structures in Computer Science 22, p. 719-722 (2012).
4. 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).
5. Jean-Guillaume Dumas, Dominique Duval, Jean-Claude Reynaud. Cartesian effect categories are Freyd-categories. Journal of Symbolic Computation 46, p. 272-293 (2011).
6. Jean-Guillaume Dumas, Frank Heckenbach, B. David Saunders, and Volkmar Welker. Computing simplicial homology based on efficient Smith normal form algorithms. In Michael Joswig and Nobuki Takayama, editors, *Algebra, Geometry and Software Systems*, pages 177–206. Springer, March 2003.
7. Jean-Guillaume Dumas, B. David Saunders, and Gilles Villard. On efficient sparse integer matrix Smith normal form computations. *Journal of Symbolic Computation*, 32(1/2):71–99, July–August 2001.
8. Dominique Duval. Simultaneous computations in fields of different characteristics. In Erich Kaltofen and Stephen M. Watt, editors, *Computers and Mathematics*, pages 321–326. Springer US, 1989.
9. James Gosling, Bill Joy, Guy Steele, Gilad Bracha. The Java Language Specification, Third Edition. Addison-Wesley Longman (2005).
10. Bart Jacobs. A Formalisation of Java's Exception Mechanism. ESOP 2001. Springer Lecture Notes in Computer Science 2028. p. 284-301 (2001).

11. John M. Lucassen, David K. Gifford. Polymorphic effect systems. POPL 1988. ACM Press, p. 47-57.
12. 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).
13. Paul Blain Levy. Monads and adjunctions for global exceptions. MFPS 2006. Electronic Notes in Theoretical Computer Science 158, p. 261-287 (2006).
14. Eugenio Moggi. Notions of Computation and Monads. Information and Computation 93(1), p. 55-92 (1991).
15. Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, Lars Birkedal. Ynot: Dependent Types for Imperative Programs. Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08), p. 229-240 (2008).
16. Gordon D. Plotkin, John Power. Notions of Computation Determine Monads. FoSSaCS 2002. Springer-Verlag Lecture Notes in Computer Science 2303, p. 342-356 (2002).
17. Gordon D. Plotkin, John Power. Algebraic Operations and Generic Effects. Applied Categorical Structures 11(1), p. 69-94 (2003).
18. Gordon D. Plotkin, Matija Pretnar. Handlers of Algebraic Effects. ESOP 2009. Springer-Verlag Lecture Notes in Computer Science 5502, p. 80-94 (2009).
19. 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).
20. Philip Wadler. The Marriage of Effects and Monads. ICFP 1998. ACM Press, p. 63-74 (1998).