Certified proofs in programs involving exceptions
Jean-Guillaume Dumas, Dominique Duval, Burak Ekici, Jean-Claude Reynaud

To cite this version:


HAL Id: hal-00867237
https://hal.archives-ouvertes.fr/hal-00867237v3
Submitted on 13 Mar 2014

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

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Certified proofs in programs involving exceptions

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

March 13, 2014

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 treatment 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 \([f] : \llbracket X \rrbracket \to \llbracket Y \rrbracket\) (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 \([f] : \llbracket X \rrbracket \to \llbracket Y \rrbracket + \text{Exc}\) where \(\text{Exc}\) 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 \((X \times \text{St})\text{St}\) (where \(\text{St}\) is the set of states) also provides a classification of expressions [14]; indeed, both approaches are related [19]. Lawvere theories were proposed for dealing with the operations and equations related to computational effects [15, 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 comonad of states \(X \times \text{St}\) 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 \(X + \text{Exc}\) (where \(\text{Exc}\) 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 [16, 17]. This issue has been circumvented in [18] in order to get a Hoare logic for exceptions, in [13] by using both algebras and coalgebras, and in [17] 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 ex-
pressions 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 5.5). 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, which 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 encountered, 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.

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).
### 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;
}

Figure 1: Throwing an exception upon division by a non unit

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

Figure 2: Re-throwing an exception to forward rank and iteration number information

3. Then it is sufficient to wrap the rank algorithm with a dynamic evaluation, splitting the continuation modulo both factors, see Fig. 3.

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.
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);
    }
}

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

The basic part of the syntax is a signature $\text{Sig}_{\text{base}}$, made of a types (or sorts) and operations.

The exceptional types form a subset $\mathcal{T}$ of the set of types of $\text{Sig}_{\text{base}}$. 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 $\text{Sig}_{\text{base}}$ and some set of exceptional types $\mathcal{T}$ have been chosen.

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

**Definition 3.1.** The **signature for exceptions** $\text{Sig}_{\text{exc}}$ is made of $\text{Sig}_{\text{base}}$ with the following operations: a raising operation for each exceptional type $T$ and each type $Y$:

$$\text{throw}_{T,Y}: T \rightarrow Y,$$

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

$$\text{try}\{f\} \text{ catch }\{T_i \Rightarrow g_i\}_{1 \leq i \leq n}: X \rightarrow Y.$$
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 $\text{Sig}_{\text{exc}}$, but the interpretations of the operations for raising and handling exceptions, which are part of $\text{Sig}_{\text{exc}}$, are defined in terms of the interpretations of the core operations. The meaning of the core operations is given in Section 4.

**Definition 3.2.** Let $\text{Sig}_{\text{exc}}$ be the signature for exceptions. The core of $\text{Sig}_{\text{exc}}$ is the signature $\text{Sig}_{\text{core}}$ made of $\text{Sig}_{\text{base}}$ with a type $\emptyset$ called the empty type and two operations for each exceptional type $T$: 

\[ \text{tag}_T : T \rightarrow \emptyset \quad \text{and} \quad \text{untag}_T : \emptyset \rightarrow T \]

where $\text{tag}_T$ is called the exception constructor or the tagging operation and $\text{untag}_T$ 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 $\llbracket X \rrbracket$ and each operation $f : X \rightarrow Y$ of $\text{Sig}_{\text{base}}$ as a function $\llbracket f \rrbracket : \llbracket X \rrbracket \rightarrow \llbracket Y \rrbracket$.

But, when $f : X \rightarrow Y$ in $\text{Sig}_{\text{exc}}$ is a raising or handling operation, or when $f : X \rightarrow Y$ in $\text{Sig}_{\text{core}}$ is a tagging or untagging operation, it is not interpreted as a function $\llbracket f \rrbracket : \llbracket X \rrbracket \rightarrow \llbracket Y \rrbracket$: 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 $\text{Sig}_{\text{base}}$ 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
types 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 \( \text{Exc} \) be a set, called the *set of exceptions*.

For each set \( A \), the set \( A + \text{Exc} \) is the disjoint union of \( A \) and \( \text{Exc} \) and
the canonical inclusions are denoted \( \text{normal}_A : A \to A + \text{Exc} \) and \( \text{abrupt}_A : \text{Exc} \to A + \text{Exc} \). For each functions \( f : A \to B \) and \( g : \text{Exc} \to B \), we denote
by \( [f|g] : A + \text{Exc} \to B \) the unique function such that \( [f|g] \circ \text{normal}_A = f \) and
\( [f|g] \circ \text{abrupt}_A = g \).

**Definition 4.1.** An element of \( A + \text{Exc} \) is an *ordinary value* if it is in \( A \) and an
*exceptional value* if it is in \( \text{Exc} \).

A function \( \varphi : A + \text{Exc} \to B + \text{Exc} \): \( \varphi \) raises an exception if there is some \( x \in A \) such that \( \varphi(x) \in \text{Exc} \).

\( \varphi \) recovers from an exception if there is some \( e \in \text{Exc} \) such that \( \varphi(e) \in B \).

\( \varphi \) propagates exceptions if \( \varphi(e) = e \) for every \( e \in \text{Exc} \).

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

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

**Definition 4.2.** The *upcasting conversions* are the following transformations:

- every function \( \varphi : A \to B \) gives rise to \( \uparrow \varphi = \text{normal}_B \circ \varphi : A \to B + \text{Exc} \),
- every function \( \psi : A \to B + \text{Exc} \) gives rise to \( \uparrow \psi = \llbracket \psi \rrbracket \text{abrupt}_B \) : \( A + \text{Exc} \to B + \text{Exc} \), which is equal to \( \psi \) on \( A \) and which propagates exceptions;
- it follows that every function \( \varphi : A \to B \) gives rise to \( \uparrow \varphi = \llbracket \varphi \rrbracket \text{normal}_B \circ \text{abrupt}_B = \varphi + \text{id}_{\text{Exc}} : A + \text{Exc} \to B + \text{Exc} \), which is equal to \( \varphi \) 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 \( \llbracket f \rrbracket : \llbracket X \rrbracket + \text{Exc} \to \llbracket Y \rrbracket + \text{Exc} \) and \( \llbracket g \rrbracket : \llbracket Y \rrbracket + \text{Exc} \to \llbracket Z \rrbracket + \text{Exc} \), which can be composed.
Thus, every term of $\text{Sig}_{\text{exc}}$ and $\text{Sig}_{\text{core}}$ can be interpreted by first converting the interpretation of each of its components $f : X \rightarrow Y$ to a function $\llbracket f \rrbracket : [X] + \text{Exc} \rightarrow [Y] + \text{Exc}$.

For $\text{Sig}_{\text{exc}}$, this coincides with the Kleisli composition associated to the exception monad $A + \text{Exc}$ [14].

We will also use the following conversion.

**Definition 4.3.** The downcasting conversion is the following transformation:

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

This conversion is unsafe: different $\theta$’s may give rise to the same $\theta^*$.

### 4.2 Tagging and raising exceptions

Raising exceptions relies on the interpretation of the tagging operations. The interpretation of the empty type $\emptyset$ is the empty set $\emptyset$; thus, for each type $X$ the interpretation of $\emptyset + X$ can be identified with $\llbracket X \rrbracket$.

**Definition 4.4.** Let $\text{Exc}$ 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 $\text{tag}_T : T \rightarrow \emptyset$ is the coprojection function

$$\llbracket \text{tag}_T \rrbracket : \llbracket T \rrbracket \rightarrow \text{Exc}.$$  

Thus, the tagging function $\llbracket \text{tag}_T \rrbracket : \llbracket T \rrbracket \rightarrow \text{Exc}$ maps a non-exceptional value (or parameter) $a \in \llbracket T \rrbracket$ to an exception $\llbracket \text{tag}_T \rrbracket(a) \in \text{Exc}$.

We can now define the raising of exceptions in a programming language.

**Definition 4.5.** For each exceptional type $T$ and each type $Y$, the interpretation of the raising operation $\text{throw}_{T,Y}$ is the tagging function $\llbracket \text{tag}_T \rrbracket$ followed by the inclusion of $\text{Exc}$ in $\llbracket Y \rrbracket + \text{Exc}$:

$$\llbracket \text{throw}_{T,Y} \rrbracket = \text{abrupt}_{\llbracket Y \rrbracket} \circ \llbracket \text{tag}_T \rrbracket : \llbracket T \rrbracket \rightarrow \llbracket Y \rrbracket + \text{Exc}.$$  

### 4.3 Untagging and handling exceptions

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

**Definition 4.6.** For each exceptional type $T$, the interpretation of the untagging operation $\text{untag}_T : \emptyset \rightarrow T$ is the function

$$\llbracket \text{untag}_T \rrbracket : \text{Exc} \rightarrow \llbracket T \rrbracket + \text{Exc},$$  

which satisfies for each exceptional type $R$:

$$\llbracket \text{untag}_T \rrbracket \circ \llbracket \text{tag}_R \rrbracket = \begin{cases} \text{normal}_T & \text{when } R = T \\ \text{abrupt}_T \circ \llbracket \text{tag}_R \rrbracket & \text{when } R \neq T \end{cases} : \llbracket R \rrbracket \rightarrow \llbracket T \rrbracket + \text{Exc}.$$  

8
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 4.6.

For handling exceptions of types $T_1, \ldots, T_n$, raised by the interpretation of some term $f : X \to Y$ of $Sig_{exc}$, one provides for each $i$ in $\{1, \ldots, n\}$ 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 4.7.** 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:

$$recover_i = \begin{cases} \lfloor [g_n] | abrupt_Y \rfloor \circ untag_{T_n}, & \text{when } i = n \\ \lfloor [g_i] | recover_{i+1} \rfloor \circ untag_{T_i}, & \text{when } i < n \end{cases}$$

Thus, 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 $[f] : [X] \to [Y]$.

**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 \([f] : [X] \rightarrow [Y] + \text{Exc}\), or as \([f] : [X] + \text{Exc} \rightarrow [Y] + \text{Exc}\).

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.

\[ \begin{array}{c}
\text{Signature} \\
\text{(Section 3)} \\
\rightarrow \\
\text{Decorated signature} \\
\text{(Section 5)} \\
\sim \\
\text{Semantics} \\
\text{(Section 4)} \\
\end{array} \]

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:

- (≡) a strong equation is an equality of functions both on ordinary values and on exceptions
- (∼) 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,
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Decorated syntax</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>type ( X )</td>
<td>type ( X )</td>
<td>([X])</td>
</tr>
<tr>
<td>term ( X \xrightarrow{f} Y )</td>
<td>pure term ( X \xrightarrow{f^{(0)}} Y )</td>
<td>([X] \xrightarrow{f} [Y])</td>
</tr>
<tr>
<td>term ( X \xrightarrow{f} Y )</td>
<td>propagator ( X \xrightarrow{f^{(1)}} Y )</td>
<td>([X] \xrightarrow{f} [Y] + \text{Exc})</td>
</tr>
<tr>
<td>term ( X \xrightarrow{f} Y )</td>
<td>catcher ( X \xrightarrow{f^{(2)}} Y )</td>
<td>([X] + \text{Exc} \xrightarrow{f} [Y] + \text{Exc})</td>
</tr>
<tr>
<td>equation ( f = g : X \rightarrow Y )</td>
<td>strong equation ( f^{(2)} \equiv g^{(2)} : X \rightarrow Y )</td>
<td>([f] = [g])</td>
</tr>
<tr>
<td>equation ( f = g : X \rightarrow Y )</td>
<td>weak equation ( f^{(2)} \sim g^{(2)} : X \rightarrow Y )</td>
<td>([f] \circ \text{normal}<em>{[X]} = [g] \circ \text{normal}</em>{[X]})</td>
</tr>
</tbody>
</table>

Figure 4: Interpretation of the decorated syntax.

so that it is not a restriction to give the interpretation of the decorated equations only when both members are catchers.

Now we can add decorations to the signature for exceptions \(\text{Sig}_{\text{exc}}\) and its associated core signature \(\text{Sig}_{\text{core}}\), from Definitions 3.1 and 3.2.

**Definition 5.1.** The decorated signature for exceptions \(\text{Sig}_{\text{exc}}^{\text{deco}}\) and its associated decorated core signature \(\text{Sig}_{\text{core}}^{\text{deco}}\) are made of \(\text{Sig}_{\text{exc}}\) and \(\text{Sig}_{\text{core}}\), 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, “\(f^{(3)} \sim g^{(1)}\)” 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: \(\frac{H_1, \ldots, H_n}{C_1, \ldots, C_p}\) stands for \(\frac{H_1}{C_1}, \ldots, \frac{H_n}{C_p}\).
(a) Monadic equational rules for exceptions (first part):

\[
\begin{array}{llllllllllllll}
& f : X \to Y & g : Y \to Z & X & f & f \equiv g & f \equiv g & g \equiv h \\
g \circ f : X \to Z & id_X : X \to X & f \equiv f & g \equiv f & f \equiv h \\
f : X \to Y & g_1 \equiv g_2 : Y \to Z & f_1 \equiv f_2 : X \to Y & g : Y \to Z & g \circ f_1 \equiv g \circ f_2 : X \to Z \\
f : X \to Y & g : Y \to Z & h : Z \to W & h \circ (g \circ f) \equiv (h \circ g) \circ f & f \circ id_X \equiv f & id_Y \circ f \equiv f \\
\end{array}
\]

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

\[
\begin{array}{llllllllllllll}
f^{(0)} & f^{(1)} & (g \circ f)^{(0)} & f^{(0)} & g^{(0)} & (g \circ f)^{(0)} & f^{(1)} & g^{(1)} & f^{(1)} \sim g^{(1)} & f \equiv g & f \sim g \\
f & f \sim f & g \sim f & f \sim g & g \sim h & f \sim h \\
f^{(0)} : X \to Y & g_1 \sim g_2 : Y \to Z & f_1 \sim f_2 : X \to Y & g : Y \to Z & g \circ f_1 \sim g \circ f_2 : X \to Z \\
g_1 \circ f \sim g_2 \circ f & g_1 \sim g_2 : Y \to Z & g \circ f_1 \sim g \circ f_2 : X \to Z \\
\end{array}
\]

(c) Rules for the empty type 0:

\[
\begin{array}{llllllllllllll}
X \to 0 & \to X & f, g : 0 \to Y & f \sim g \\
\end{array}
\]

(d) Case distinction with respect to \( X + 0 \):

\[
\begin{array}{llllllllllllll}
g^{(1)} : X \to Y & k^{(2)} : 0 \to Y \\
[g \mid k]^{(2)} : X \to Y & [g \mid k] \sim g & [g \mid k] \circ \emptyset \equiv k \\
f, g : X \to Y & f \sim g & f \circ \emptyset \equiv g \circ \emptyset \\
f \equiv g \\
\end{array}
\]

(e) Propagating exceptions:

\[
\begin{array}{llllllllllllll}
k^{(2)} : X \to Y \\
(k^{(2)})^{(1)} : X \to Y & \downarrow^k \sim k \\
\end{array}
\]

(f) Tagging:

\[
\begin{array}{llllllllllllll}
T \in \mathcal{T} & \to 0 \\
(tag_T)^{(1)} : T \to \emptyset & (f_T^{(1)} : T \to Y)_{T \in \mathcal{T}} \\
[f_T]^{(2)}_{T \in \mathcal{T}} : \emptyset \to Y & [f_T]_{T \in \mathcal{T}} \circ \tag_T \sim f_T \\
f, g : 0 \to Y & f \circ \tag_T \sim g \circ \tag_T \text{ for all } T \in \mathcal{T} \\
f \equiv g \\
\end{array}
\]

(g) Untagging:

\[
\begin{array}{llllllllllllll}
T \in \mathcal{T} & \to 0 & \to T & R, T \in \mathcal{T} & R \neq T \\
untag_T^{(2)} : \emptyset \to T & \text{untag}_T \circ \tag_T \sim id_T & \text{untag}_T \circ \tag_R \sim \emptyset \\
\end{array}
\]

Figure 5: Decorated rules for exceptions
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 “0” side of \( X + 0 \)) and non-exceptional arguments (on the “\( X \)” side of \( X + 0 \)).

The symbol \( \top \) in rules (e) is interpreted as the downcast conversion, see Definition 4.3.

Rules in (f) mean that the tagging operations are interpreted as the canonical inclusions of the exceptional types in the set \( \text{Exc} \), see Definition 4.4.

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 4.6.

Remark 5.2. 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 5.4) for constructing the raising and handling operations from the core tagging and untagging operations. It has to be noted that the term \( \text{catch} \{ T_i \Rightarrow g_i \}_1 \leq i \leq n \circ f \) may catch exceptions, while the handling operation, which coincides with \( \text{catch} \{ T_i \Rightarrow g_i \}_1 \leq i \leq n \circ f \) on non-exceptional values, must propagate exceptions; this is why the downcast operator \( \top \) is used.

**Definition 5.3.** For each exceptional type \( T \) and each type \( Y \), the raising operation \( \text{throw}^{(1)}_{T,Y} : T \rightarrow Y \) is the propagator defined as:

\[
\text{throw}^{(1)}_{T,Y} = [\mid Y \circ \text{tag}_T].
\]

**Definition 5.4.** For each propagator \( f^{(1)} : X \rightarrow Y \), each non-empty lists of types \( (T_i)_{1 \leq i \leq n} \) and propagators \( (g^{(1)}_i : T_i \rightarrow Y)_{1 \leq i \leq n} \), let \( \text{catch} \{ T_i \Rightarrow g^{(1)}_i \}_1 \leq i \leq n \circ f \) denote the catcher defined by:

\[
\text{catch} \{ T_i \Rightarrow g^{(1)}_i \}_1 \leq i \leq n = [ \mid Y \mid \text{recover}_1]
\]

where \( (\text{recover}^{(2)}_i : 0 \rightarrow Y)_{1 \leq i \leq n} \) denotes the family of catchers defined recursively by:

\[
\text{recover}^{(2)}_i = \begin{cases} g_n \circ \text{untag}_{T_n} & \text{when } i = n, \\ [g_i \mid \text{recover}_{i+1}] \circ \text{untag}_{T_i} & \text{when } i < n. \end{cases}
\]
Then, the *handling* operation $(\text{try}\{f\} \text{catch}\{T_i \Rightarrow g_i\}_{1 \leq i \leq n})^{(1)} : X \rightarrow Y$ is the propagator defined by:

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

When $n = 1$ we get:

$$\text{try}\{f\} \text{catch}\{T \Rightarrow g\} = ^i (\text{catch}\{T \Rightarrow g\} \circ f) = ^i ([ \text{id}_Y \mid g \circ \text{untag}_T ] \circ f).$$

Now Theorem 5.5 derives easily, by induction, from Fig. 5 and 4 and from Definitions 5.3 and 5.4.

**Theorem 5.5.** 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 $\llbracket Z \rrbracket$ (or, more precisely, of $\llbracket \cdot \rrbracket Z$) is $\text{abrupt}[Z] : \text{Exc} \rightarrow [Z] + \text{Exc}$.

**Lemma 5.6.** For each propagator $g^{(1)} : X \rightarrow Y$ 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.

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. Last, the application of the *from_empty_is_weakly_unique* rule certifies that the term $g \circ (@\text{empty} \: X)$, because its domain is the empty type, is weakly equal to the term $@\text{empty} \: Y$. Thus, we have the left side sub-proof: $g \circ (@\text{empty} \: X) \sim (@\text{empty} \: Y)$. There, weak equality is converted into strong: applying *propagator_weakeq_is_strongeq* resolves the goal: $g \circ (@\text{empty} \: X) \equiv (@\text{empty} \: Y)$ 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)1 *) 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.

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 \( T \) is endowed with a partial order \( \rightarrowtriangle \) called the subtyping relation, and the signature \( \text{Sig}_\text{base} \) is extended with a cast operation
\[
\text{cast}_{R,T} : R \rightarrow T
\]
whenever \( R \rightarrowtriangle T \).

The interpretation of \( \text{cast}_{R,T} \) is a pure function
\[
\llbracket \text{cast}_{R,T} \rrbracket : \llbracket R \rrbracket \rightarrow \llbracket T \rrbracket,
\]
such that \( \llbracket \text{cast}_{T,T} \rrbracket \) is the identity on \( \llbracket T \rrbracket \) and when \( S \rightarrow R \rightarrowtriangle T \) then
\[
\llbracket \text{cast}_{S,T} \rrbracket = \llbracket \text{cast}_{R,T} \rrbracket \circ \llbracket \text{cast}_{S,R} \rrbracket.
\]

Definition 4.6 has to be modified as follows: the function
\[
\llbracket \text{untag}_T \rrbracket : \text{Exc} \rightarrow \llbracket T \rrbracket + \text{Exc}
\]
satisfies for each exceptional type \( R \):
\[
\llbracket \text{untag}_T \rrbracket \circ \llbracket \text{tag}_R \rrbracket =
\begin{cases}
\llbracket \text{normal}_T \rrbracket \circ \llbracket \text{cast}_{R,T} \rrbracket & \text{when } R \rightarrow T \\
\llbracket \text{abrupt}_T \rrbracket \circ \llbracket \text{tag}_R \rrbracket & \text{otherwise}
\end{cases}
\]

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 3.1 and 3.2) 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 languages.

References


