Adjunctions for exceptions

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

Adjunctions for exceptions

Jean-Guillaume Dumas [*], Dominique Duval [†], Laurent Fousse, Jean-Claude Reynaud
Université de Grenoble, Laboratoire Jean Kuntzmann
{jgdumas,dduval,lfousse,jcreynaud}@imag.fr

October 26, 2012

Abstract

Abstract. The exceptions form a computational effect, in the sense that there is an apparent mismatch between the syntax of exceptions and their intended semantics. We solve this apparent contradiction by defining a logic for exceptions with a proof system which is close to their syntax and where their intended semantics can be seen as a model. This requires a robust framework for logics and their morphisms, which is provided by diagrammatic logics.

Keywords. Computational effects. Semantics of exceptions. Adjunction. Categorical fractions. Limit sketches. Diagrammatic logics. Morphisms of logics. Decorated proof system.

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 $f : X \to Y + E$ where E is the set of exceptions. In a computer language usually exceptions 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. Besides, the theory of *diagrammatic logics* forms a new paradigm for understanding the nature of computational effects; in this paper, diagrammatic logics are applied to the denotational semantics of exceptions.

To our knowledge, the first categorical treatment of computational effects is due to Moggi [15]; this approach relies on *monads*, it is implemented in the programming language Haskell [21, 12]. The examples proposed by Moggi include the states monad $TX = (X \times S)^S$ where S is the set of states and the exceptions monad $TX = X + E$ where E is the set of exceptions. Later on, using the correspondence between monads and algebraic theories, Plotkin and Power proposed to use *Lawvere theories* for dealing with the operations and equations related to computational effects, for instance the lookup and update operations for states and the raising and handling operations for exceptions [16, 13]. In the framework of Lawvere theories, an operation is called *algebraic* when it satisfies some relevant genericity properties; the operations lookup and update for states and the operation for raising exceptions are algebraic in this sense, while the operation for handling exceptions is not [17]. This difficulty can be overcome, as for instance in [20, 14, 18]. Nevertheless, from these points of view the handling of exceptions is inherently different from the updating of states.

In this paper we use another method for dealing with computational effects. This method has been applied to a parametrization process in [1, 2] and to the states effect in [4]. It has led to the discovery of a duality between states and exceptions, briefly presented in [3]. Our approach also provides a notion of *sequential product*, which is an alternative to the strength of a monad for imposing an evaluation order for the arguments of a n-ary function [5]. With this point of view the fact that the handling operation for exceptions is not algebraic, in the sense of Lawvere theories, is not an issue. In fact, the duality between the

[*]This work is partly funded by the HPAC project of the French Agence Nationale de la Recherche (ANR 11 BS02 013).
[†]This work is partly funded by the CLIMT project of the French Agence Nationale de la Recherche (ANR 11 BS02 016).

exceptions effect and the states effect [3] implies that catching an exception is dual to updating a state. 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"), which encapsulates the catching operation.

Our idea is to look at an effect as an apparent mismatch between syntax and semantics: there is one logic which fits with the syntax, another one which fits with the semantics, and a third one which reconciles syntax and semantics. This third logic classifies the language features and their properties according to the way they interact with the effect; we call this kind of classification a *decoration.* For this conciliation, as the features of the different logics are quite different in nature, we will use morphisms from the decorated logic to each of the two other logics, in a relevant category.

This approach thus requires a robust framework for dealing with logics and morphisms between them. This is provided by the category of *diagrammatic logics* [6, 1]. The main ingredient for defining this category is the notion of categorical *fraction*, as introduced in [9] for dealing with homotopy theory. Fractions are defined with respect to an *adjunction.* The syntactic aspect of logics is obtained by assuming that this adjunction is induced by a morphism of *limit sketches* [7], which implies that the adjunction connects *locally presentable* categories. For each diagrammatic logic we define *models* as relevant morphisms, *inference rules* as fractions and *inference steps* as composition of fractions. Thus, diagrammatic logics are defined from well-known categorical ingredients; their novelty lies in the importance given to fractions, in the categorical sense, for formalizing logics.

The category of diagrammatic logics is introduced in Section 1. In Section 2 we look at exceptions from an *explicit* point of view, by introducing a type of exceptions in the return type of operations which may raise exceptions. With this explicit point of view we formalize (by Definition 2.13) the intended semantics of exceptions as provided in the documentation of the computer language Java [10] and reminded in Appendix A. We also introduce the distinction between the core operations and their encapsulation: typically, between the catching and the handling of exceptions. This explicit point of view is expressed in terms of a diagrammatic logic denoted $\mathcal{L}_{expl}$: the intended semantics of exceptions can be seen as a model with respect to $\mathcal{L}_{expl}$. Then in Section 3 we look at exceptions from a *decorated* point of view, which fits with the syntax much better than the explicit point of view, since the return type of operations does not mention any type of exceptions. The key point in this logic is that the operations and equations are decorated according to their interaction with exceptions. This decorated point of view corresponds to another diagrammatic logic denoted $\mathcal{L}_{deco}$. We build a morphism of diagrammatic logics from $\mathcal{L}_{deco}$ to $\mathcal{L}_{expl}$, called the *expansion*, from which our main result (Theorem 3.15) follows: the intended semantics of exceptions can also be seen as a model with respect to $\mathcal{L}_{deco}$.

$$\begin{tikzcd}
\mathcal{L}_{deco} \arrow[r, "\text{expansion}"] \arrow[d, squiggly, "\text{model (Section 3)}"'] & \mathcal{L}_{expl} \arrow[d, squiggly, "\text{model (Section 2)}"] \\
\text{semantics} \arrow[r, equal, "\text{(Theorem 3.15)}"'] & \text{semantics}
\end{tikzcd}$$

Then we prove some properties of exceptions using the rules of the decorated logic and the duality between exceptions and states. We conclude in Section 4 with some remarks and guidelines for future work.

1 The category of diagrammatic logics

This paper relies on the robust algebraic framework provided by the category of diagrammatic logics [1, 6]. In Section 1.1 we provide an informal description of diagrammatic logics which should be sufficient for understanding most of Sections 2 and 3. Let us also mention the paper [2] for a detailed presentation of a simple application of diagrammatic logics. Precise definitions of diagrammatic logics and their morphisms are given in Section 1.2; these definitions rely on the categorical notions of fractions and limit sketches.

1.1 A diagrammatic logic is a left adjoint functor

In this Section we give an informal description of diagrammatic logics and their morphisms; the formal definitions will be given in Section 1.2. In order to define a diagrammatic logic we have to distinguish its *theories*, which are closed under deduction, from its *specifications*, which are presentations of theories. On the one hand, each specification generates a theory, by applying the *inference rules* of the logic: the specification is a family of axioms and the theory is the family of theorems which can be proved from these axioms, using the inference system of the logic. On the other hand, each theory can be seen as a ("large") specification.

Then, clearly, a *morphism* of logics has to map specifications to specifications and theories to theories, in some consistent way. The diagrammatic logics we are considering in this paper are variants of the equational logic: their specifications are made of (some kinds of) sorts, operations and equations. Each sort, operation or equation can be seen as a specification, hence every morphism of logics has to map it to a specification. However it is not required that a sort be mapped to a sort, an operation to an operation or an equation to an equation. Thanks to this property, rather subtle relations between logics can be formalized by morphisms of diagrammatic logics. This is the case for the expansion morphism, see Figure 3.

A diagrammatic logic is a left adjoint functor $\mathcal{L}$ from a category $\mathbf{S}$ of specifications to a category $\mathbf{T}$ of theories, with additional properties that will be given in Section 1.2. Each specification generates a theory thanks to this functor $\mathcal{L}$ and each theory can be seen as a specification thanks to the right adjoint functor $\mathcal{R} : \mathbf{T} \to \mathbf{S}$. In addition, it is assumed that the canonical morphism $\varepsilon_\Theta : \mathcal{L}\mathcal{R}\Theta \to \Theta$ is an isomorphism in $\mathbf{T}$, so that each theory Θ can be seen as a presentation of itself. The fact that indeed the functor $\mathcal{L}$ describes an *inference system* is due to additional assumptions on the adjunction $\mathcal{L} \dashv \mathcal{R}$, which are given in the next Section.

Although this point will not be formalized, in order to understand the definition of the models of a specification it may be helpful to consider that one is usually interested in two kinds of theories: the theories $\mathcal{L}\Sigma$ which are generated by a "small" (often finite) specification Σ, and the "large" theories Θ which are provided by set theory, domain theory and other mathematical means, to be used as interpretation domains. However, formally this distinction is useless, and the *models* of any specification Σ with values in any theory Θ are defined as the morphisms from $\mathcal{L}\Sigma$ to Θ in $\mathbf{T}$. Thanks to the adjunction $\mathcal{L} \dashv \mathcal{R}$, there is an alternative definition which has a more constructive flavour: the *models* of Σ with values in Θ are the morphisms from Σ to $\mathcal{R}\Theta$ in $\mathbf{S}$.

The definition of *morphisms* between diagrammatic logics derives in an obvious way from the definition of diagrammatic logics: a morphism $F : \mathcal{L}_1 \to \mathcal{L}_2$ is made of two functors $F_S : \mathbf{S}_1 \to \mathbf{S}_2$ and $F_T : \mathbf{T}_1 \to \mathbf{T}_2$ with relevant properties.

In this paper we consider several diagrammatic logics which are variants of the equational logic: the specifications are defined from sorts, operations and equations, and the inference rules are variants of the usual equational rules. Exceptions form a *computational effect*, in the sense that a syntactic expression $f : X \to Y$ may be interpreted as a function $f : X \to Y + E$ (where E is the set of exceptions) instead of being interpreted as a function $f : X \to Y$. We will define a logic $\mathcal{L}_{deco}$ for dealing with the syntactic expressions and another logic $\mathcal{L}_{expl}$ for dealing with exceptions in an explicit way by adding a sort of exceptions (also denoted E). The key feature of this paper is the *expansion* morphism form the logic $\mathcal{L}_{deco}$ to the logic $\mathcal{L}_{expl}$, which maps a syntactic expression $f : X \to Y$ to the expression $f : X \to Y + E$ in a consistent way.

1.2 Diagrammatic logics, categorically

The notion of diagrammatic logic is an algebraic notion which captures some major properties of logics and which provides a simple and powerful notion of morphism between logics. Each diagrammatic logic comes with a notion of models and it has a sound inference system.

A category is *locally presentable* when it is equivalent to the category $\mathit{Real}(\mathbf{E})$ of set-valued models, or *realizations*, of a limit sketch $\mathbf{E}$ [7, 8]. The category $\mathit{Real}(\mathbf{E})$ has colimits and there is a canonical contravariant functor $\mathcal{Y}$ from $\mathbf{E}$ to $\mathit{Real}(\mathbf{E})$, called the *contravariant Yoneda functor* associated with $\mathbf{E}$, such that $\mathcal{Y}(\mathbf{E})$ generates $\mathit{Real}(\mathbf{E})$ under colimits, in the sense that every object of $\mathit{Real}(\mathbf{E})$ may be written as a colimit over a diagram with objects in $\mathcal{Y}(\mathbf{E})$.

Each morphism of limit sketches $\mathbf{e} : \mathbf{E} \to \mathbf{E}'$ gives rise, by precomposition with $\mathbf{e}$, to a functor $G_{\mathbf{e}} : \mathit{Real}(\mathbf{E}') \to \mathit{Real}(\mathbf{E})$, which has a left adjoint $F_{\mathbf{e}}$ [7]. Let $\mathcal{Y}$ and $\mathcal{Y}'$ denote the contravariant Yoneda functors associated with $\mathbf{E}$ and $\mathbf{E}'$, respectively. Then $F_{\mathbf{e}}$ extends $\mathbf{e}$, which means that $F_{\mathbf{e}} \circ \mathcal{Y} = \mathcal{Y}' \circ \mathbf{e}$ up to a natural isomorphism. We call such a functor $F_{\mathbf{e}}$ a *locally presentable functor*. Then the three following properties are equivalent: the counit $\varepsilon : F_{\mathbf{e}} \circ G_{\mathbf{e}} \Rightarrow \mathit{Id}$ is a natural isomorphism; the right adjoint $G_{\mathbf{e}}$ is full and faithful; the left adjoint $F_{\mathbf{e}}$ is (up to an equivalence of categories) a *localization*, i.e., it consists of adding inverses to some morphisms from $\mathit{Real}(\mathbf{E})$, constraining them to become isomorphisms in $\mathit{Real}(\mathbf{E}')$ [9]. Then it can be assumed that $\mathbf{e}$ is also a localization: it consists of adding inverses to some morphisms from $\mathbf{E}$.

Definition 1.1. A *diagrammatic logic* is a locally presentable functor which is a localization, up to an equivalence of categories.

It follows that a diagrammatic logic is a left adjoint functor such that its counit is a natural isomorphism: these properties have been used in Section 1.1.

Definition 1.2. Let $\mathcal{L} : \mathbf{S} \to \mathbf{T}$ be a diagrammatic logic with right adjoint $\mathcal{R}$.

- The category of $\mathcal{L}$-*specifications* is $\mathbf{S}$.
- The category of $\mathcal{L}$-*theories* is $\mathbf{T}$.
- A *model* of a specification Σ with values in a theory Θ is a morphism from $\mathcal{L}\Sigma$ to Θ in $\mathbf{T}$, or equivalently (thanks to the adjunction) a morphism from Σ to $\mathcal{R}\Theta$ in $\mathbf{S}$.

The *bicategory of fractions* associated to $\mathcal{L}$ has the same objects as $\mathbf{S}$ and a morphism from Σ_1 to Σ_2 in this bicategory is a *fraction* $\tau\backslash\sigma : \Sigma_1 \to \Sigma_2$, which means that it is a cospan $(\sigma : \Sigma_1 \to \Sigma'_2 \leftarrow \Sigma_2 : \tau)$ in $\mathbf{S}$ such that $\mathcal{L}\tau$ is invertible in $\mathbf{T}$. Then σ is called the *numerator* and τ the *denominator* of the fraction $\tau\backslash\sigma$. It follows that we can define $\mathcal{L}(\tau\backslash\sigma) = \mathcal{L}\tau^{-1} \circ \mathcal{L}\sigma$. The composition of consecutive fractions is defined as the composition of cospans, using a pushout in $\mathbf{S}$.

Definition 1.3. Let $\mathcal{L} : \mathbf{S} \to \mathbf{T}$ be a diagrammatic logic with right adjoint $\mathcal{R}$.

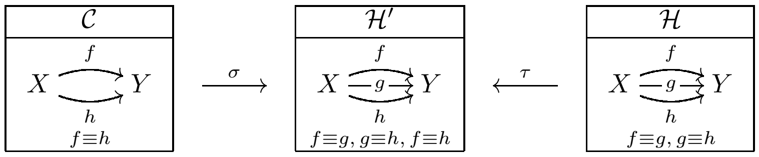
- A *rule* with *hypothesis* $\mathcal{H}$ and *conclusion* $\mathcal{C}$ is a fraction from $\mathcal{C}$ to $\mathcal{H}$ with respect to $\mathcal{L}$.
- An *instance* of a specification Σ_0 in a specification Σ is a fraction from Σ_0 to Σ with respect to $\mathcal{L}$.
- The *inference step* applying a rule $\rho : \mathcal{C} \to \mathcal{H}$ to an instance $\iota : \mathcal{H} \to \Sigma$ of $\mathcal{H}$ in Σ is the composition of fractions $\iota \circ \rho : \mathcal{C} \to \Sigma$; it yields an instance of $\mathcal{C}$ in Σ.

Definition 1.4. Let $\mathcal{L} : \mathbf{S} \to \mathbf{T}$ be a diagrammatic logic with right adjoint $\mathcal{R}$.

- Each morphism of limit sketches $\mathbf{e} : \mathbf{E}_S \to \mathbf{E}_T$ which gives rise to the adjunction $\mathcal{L} \dashv \mathcal{R}$ and which is a localization is called an *inference system* for $\mathcal{L}$.
- Then a rule $\tau\backslash\sigma$ is *elementary* if σ and τ are the images, by the canonical contravariant functor $\mathcal{Y}$, of arrows s and t in $\mathbf{E}_S$ such that $\mathbf{e}(t)$ is invertible in $\mathbf{E}_T$; otherwise the rule $\tau\backslash\sigma$ is *derivable*.

Remark 1.5. An inference rule is usually written as a fraction $\frac{\mathcal{H}_1 \dots \mathcal{H}_k}{\mathcal{C}}$, it is indeed related to a categorical fraction, as follows (however from the categorical point of view the numerator is on the conclusion side and the denominator on the hypothesis side!). First let us remark that each $\mathcal{H}_i$ can be seen as a specification, as well as $\mathcal{C}$, and that the common parts in the $\mathcal{H}_i$'s and in $\mathcal{C}$ are indicated by using the same names. Then let $\mathcal{H}$ be the vertex of the colimit of the $\mathcal{H}_i$'s, amalgamated according to their common names. The fraction $(\sigma : \mathcal{C} \to \mathcal{H}' \leftarrow \mathcal{H} : \tau)$ is defined as the pushout of $\mathcal{H}$ and $\mathcal{C}$ over their common names. Then the rule $\frac{\mathcal{H}_1 \dots \mathcal{H}_k}{\mathcal{C}}$ corresponds to the categorical fraction $\tau\backslash\sigma : \mathcal{C} \to \mathcal{H}$ (see Example 1.6). In an inference system $\mathbf{e} : \mathbf{E}_S \to \mathbf{E}_T$ for a logic $\mathcal{L}$, the limit sketch $\mathbf{E}_S$ describes the syntax and the morphism $\mathbf{e}$ provides the inference rules of $\mathcal{L}$. Thus, the description of a diagrammatic logic via one of its inference systems can be done algebraically by defining $\mathbf{e}$ or the image of $\mathbf{e}$ by the canonical funtor $\mathcal{Y}$ (examples can be found in [2]). A diagrammatic logic can also be defined more traditionally by giving a grammar and a family of rules. Moreover, when the logic is simple enough, it may be sufficient in practice to describe its theories.

Example 1.6 (Monadic equational logic)**.** The monadic equational logic $\mathcal{L}_{meq}$ can be defined from its theories. A monadic equational theory is a category where the axioms hold only up to some congruence relation. Precisely, a *monadic equational theory* is a directed graph (its vertices are called *types* and its edges are called *terms*) with an *identity* term $id_X : X \to X$ for each type X and a *composed* term $g \circ f : X \to Z$ for each pair of consecutive terms $(f : X \to Y, g : Y \to Z)$; in addition it is endowed with *equations* $f \equiv g : X \to Y$ that form an equivalence relation on parallel terms which is a *congruence* with respect to the composition and such that the associativity and identity axioms hold up to congruence. The category of sets forms a $\mathcal{L}_{meq}$-theory **Set** where types, terms and equations are the sets, functions and equalities. We can look at a rule, for instance the transitivity rule for equations $\frac{f\equiv g \quad g\equiv h}{f\equiv h}$, as a categorical fraction $\tau\backslash\sigma : \mathcal{C} \to \mathcal{H}$, as follows.



Remark 1.7. Diagrammatic logics generalize *E-doctrines*, in the sense of [22]. Let E be a type of sketch, determined by what sorts of cones and cocones are allowed in the sketch. Then E determines a type of category, required to have all (co)limits of the sorts of (co)cones allowed by E, and it determines a type of functor, required to preserve that sorts of (co)limits. Following [22], the *E-doctrine* is made of these sketches, categories and functors. Each E-doctrine corresponds to a diagrammatic logic $\mathcal{L}_E : \mathbf{S}_E \to \mathbf{T}_E$, where $\mathbf{S}_E$ is the category of E-sketches (with the morphisms of E-sketches), $\mathbf{T}_E$ is the category of E-categories and E-functors, and $\mathcal{L}_E$ is the left adjoint functor which maps each E-sketch to its *theory*. For instance the E-doctrine made of finite products sketches, cartesian categories and functors preserving finite products corresponds to the equational logic.

An important feature of diagrammatic logics is their simple and powerful notion of morphism, which is a variation of the notion of morphism in an arrow category.

Definition 1.8. Given diagrammatic logics $\mathcal{L} : \mathbf{S} \to \mathbf{T}$ and $\mathcal{L}' : \mathbf{S}' \to \mathbf{T}'$, a *morphism of diagrammatic logics* $\mathcal{F} : \mathcal{L} \to \mathcal{L}'$ is made of two locally presentable functors $\mathcal{F}_S : \mathbf{S} \to \mathbf{S}'$ and $\mathcal{F}_T : \mathbf{T} \to \mathbf{T}'$ such that the square of left adjoints $(\mathcal{L}, \mathcal{L}', \mathcal{F}_S, \mathcal{F}_T)$ is induced by a commutative square of limit sketches. It follows that the right adjoints form a commutative square and that the left adjoints form a square which is commutative up to a natural isomorphim.

This means that a morphism from $\mathcal{L}$ to $\mathcal{L}'$ maps (in a coherent way) each specification of $\mathcal{L}$ to a specification of $\mathcal{L}'$ and each proof of $\mathcal{L}$ to a proof of $\mathcal{L}'$. Moreover, it is sufficient to check that each elementary specification (i.e., each specification in the image of the functor $\mathcal{Y}$) of $\mathcal{L}$ is mapped to a specification of $\mathcal{L}'$ and that each elementary proof (i.e., each inference rule) of $\mathcal{L}$ is mapped to a proof of $\mathcal{L}'$. The next result is the key point for proving Theorem 3.15; its proof is a straightforward application of the properties of adjunctions.

Proposition 1.9. *Let $\mathcal{F} = (\mathcal{F}_S, \mathcal{F}_T) : \mathcal{L} \to \mathcal{L}'$ be a morphism of diagrammatic logics and let $\mathcal{G}_T$ be the right adjoint of $\mathcal{F}_T$. Let Σ be a $\mathcal{L}$-specification and Θ' a $\mathcal{L}'$-theory. Then there is a bijection, natural in Σ and Θ':*

$$Mod_{\mathcal{L}}(\Sigma, \mathcal{G}_T\Theta') \cong Mod_{\mathcal{L}'}(\mathcal{F}_S\Sigma, \Theta') \ .$$

2 Denotational semantics of exceptions

In this Section we define a denotational semantics of exceptions which relies on the semantics of exceptions in Java. Syntax is introduced in Section 2.1 as a signature Sig_{exc}. The fundamental distinction between ordinary and exceptional values is discussed in Section 2.2. Sections 2.3 and 2.4 are devoted to the definitions of a logic with an explicit type of exceptions and a specification Σ_{expl} for exceptions with respect to this logic. Then in Section 2.5 the denotational semantics of exceptions is defined as a model of this specification. This is extended to higher-order constructions in Section 2.6.

We often use the same notations for a feature in a signature and for its interpretation. So, the syntax of exceptions corresponds to the signature Sig_{exc}, while the semantics of exceptions is defined as a model of a specification Σ_{expl}. But the signature underlying Σ_{expl} is different form Sig_{exc}: this mismatch is due to the fact that the exceptions form a computational effect. The whole paper can be seen as a way to reconcile both points of view. This can be visualized by Figure 1, with the signature for exceptions Sig_{exc} on one side and the specification Σ_{expl} with its model M_{expl} on the other side; the aim of Section 3 will be to fill the gap between Sig_{exc} and Σ_{expl} by introducing new features in the middle, see Figure 2.

$$\begin{tikzcd}
\textbf{syntax} & & \textbf{semantics} \\
Sig_{exc} & ? \longleftrightarrow \ ? & \Sigma_{expl} \arrow[d, "M_{expl}"] \\
 & & \Theta_{expl}
\end{tikzcd}$$

Figure 1: Syntax and semantics of exceptions

2.1 Signature for exceptions

The syntax for exceptions in computer languages depends on the language: the keywords for raising exceptions may be either `raise` or `throw`, and for handling exceptions they may be either `handle`, `try-with` or `try-catch`, for instance. In this paper we rather use `throw` and `try-catch`. More precisely, the syntax of our language may be described in two parts: a *pure* part and an *exceptional* part.

The pure part is a signature Sig_{pure}. The interpretation of the pure operations should neither raise nor handle exceptions. For simplicity we assume that the pure operations are either constants or unary; general n-ary operations will be mentioned in Section 4.

The signature Sig_{exc} for exceptions is made of Sig_{pure} together with the types and operations for raising and handling exceptions. In order to deal with several types of exceptions which can be parameterized, we introduce a set of indices I and for each index $i \in I$ we choose a pure type P_i called the *type of parameters* for the exceptions of index i. The new operations in Sig_{exc} are the operations for raising and handling operations, as follows.

Definition 2.1. Let Sig_{pure} be a signature. Given a set of indices I and a type P_i of Sig_{pure} for each $i \in I$, the *signature for exceptions* Sig_{exc} is made of Sig_{pure} together with, for each $i \in I$: a *raising* (or *throwing*) operation for each type Y in Sig_{pure}:

$$throw_{Y,i} : P_i \to Y \ ,$$

and a *handling* (or *try-catch*) operation for each Sig_{exc}-term $f : X \to Y$, each non-empty list of indices $(i_1, \dots, i_n)$ in I and each family of Sig_{exc}-terms $g_1 : P_{i_1} \to Y$, ..., $g_n : P_{i_n} \to Y$:

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

Remark 2.2. The precise meaning of these operations is defined in Section 2.5. Roughly speaking, relying for instance on Java see appendix A, raising an exception signals an error, which may be "catched" by an exception handler, so that the evaluation may go on along another path. For raising an exception, $throw_{Y,i}$ turns some parameter of type P_i into an exception of index i, in such a way that this exception is considered as being of type Y. For handling an exception, the evaluation of $try\{f\}\ catch\, \{i \Rightarrow g\}$ begins with the evaluation of f; if the result is not an exception then it is returned; if the result is an exception of index i then this exception is catched, which means that its parameter is recovered and g is applied to this parameter; otherwise the exception is returned, which usually produces an error message like "uncaught exception...".

The evaluation of $try\{f\}\ catch\ \{i_1 \Rightarrow g_1|\dots|i_n \Rightarrow g_n\}$ for any $n > 1$ is similar; it is checked whether the exception returned by f has index i_1 or i_2 ... or i_n in this order, so that whenever $i_j = i_k$ with $j < k$ the clause $i_k \Rightarrow g_{i_k}$ is never executed.

2.2 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*. Let us introduce a set E called the *set of exceptions*. For each set X we consider the disjoint union $X + E$. The denotational semantics of exceptions relies on the following facts. Each type X in Sig_{exc} is interpreted as a set X. Each term $f : X \to Y$ is interpreted as a function $f : X \to Y + E$, and whenever f is pure this function has its image in Y. The fact that a term $f : X \to Y$ is not always interpreted as a function $f : X \to Y$ implies that the exceptions form a *computational effect*.

Definition 2.3. For each set X, an element of $X + E$ is an *ordinary value* if it is in X and an *exceptional value* if it is in E. A function $f : X \to Y + E$ or $f : X + E \to Y + E$ *raises an exception* if there is some $x \in X$ such that $f(x) \in E$ and f *recovers from an exception* if there is some $e \in E$ such that $f(e) \in Y$. A function $f : X + E \to Y + E$ *propagates exceptions* if $f(e) = e$ for every $e \in E$.

Remark 2.4. Clearly, a function $f : X + E \to Y + E$ which propagates exceptions may raise an exception but cannot recover from an exception. Such a function f is characterized by its restriction $f|_X : X \to Y + E$. In addition, every function $f_0 : X \to Y$ can be extended in a unique way as a function $f : X + E \to Y + E$ which propagates exceptions; then $f|_X$ is the composition of f_0 with the inclusion of Y in $Y + E$.

Remark 2.5. An important feature of a language with exceptions is that the interpretation of every term is a function which propagates exceptions; *this function may raise exceptions but it cannot recover from an exception.* Indeed, the *catch* block in a *try-catch* expression may recover from exceptions which are raised inside the *try* block, but if an exception is raised before the *try-catch* expression is evaluated, this exception is propagated. Thus, the *untagging* functions that will be introduced in Section 2.3 in order to recover from exceptions are not the interpretation of any term of the signature Sig_{exc}. In fact, this is also the case for the *tagging* functions that will be used for raising exceptions. These tagging and untagging functions are called the *core* functions for exceptions; they are *private* in the sense that they do not appear in Sig_{exc}, but they are used for defining the *public* operations for raising and handling exceptions which are part of Sig_{exc}.

2.3 Explicit logic for exceptions

Let us define a logic with a type of exceptions by describing its theories.

Definition 2.6. A theory of the *explicit logic for exceptions* $\mathcal{L}_{expl}$ is a monadic equational theory (as in Example 1.6) with a distinguished type E called the *type of exceptions* and with a cocone $(normal_X : X \to X + E \leftarrow E : abrupt_X)$ for each type X, which satisfies the coproduct universal property up to congruence: for every cocone $(f : X \to Y \leftarrow E : k)$ there is a term $[f|k] : X + E \to Y$, unique up to equations, such that $[f|k] \circ normal_X \equiv f$ and $[f|k] \circ abrupt_X \equiv k$.

Definition 2.7. Let E denote a set, then $\mathbf{Set}_{E,expl}$ denotes the $\mathcal{L}_{expl}$-theory where types, terms and equations are the sets, functions and equalities, where E is the set of exceptions and where for each set X the cocone $(X \to X + E \leftarrow E)$ is the disjoint union.

Remark 2.8. In addition, it can be assumed that there is an initial type $\mathbb{0}$ (up to congruence) in each explicit theory, hence a unique term $[\,]_X : \mathbb{0} \to X$ for each type X such that the cocone $(id_X : X \to X \leftarrow \mathbb{0} : [\,]_X)$ is a coproduct up to congruence.

2.4 Explicit specification for exceptions

In order to express the meaning of the raising and handling operations we introduce new operations (called the *core* operations) and equations in such a way that the functions for raising and handling exceptions are now defined in terms of the core operations.

Definition 2.9. Let Sig_{pure} be a signature. Given a set of indices I and a type P_i in Sig_{pure} for each $i \in I$, the *explicit specification for exceptions* Σ_{expl} is the $\mathcal{L}_{expl}$-specification made of Sig_{pure} together with for each $i \in I$: an operation $t_i : P_i \to E$ called the *exception constructor* or the *tagging* operation of index i and an operation $c_i : E \to P_i + E$ called the *exception recovery* or the *untagging* function of index i, together with the equations $c_i \circ t_i \equiv normal_{P_i}$ and $c_i \circ t_j \equiv abrupt_{P_i} \circ t_j$ for all $j \neq i$. Then for each $i \in I$ the raising and handling functions are respectively defined using these two core operations as follows: the *raising* function $throw_{Y,i}$ for each type Y in Sig_{pure} is:

$$throw_{Y,i} = abrupt_Y \circ t_i : P_i \to Y + E$$

and the *handling* function:

$$try\{f\}\ catch\,\{i_1 \Rightarrow g_1 | \dots | i_n \Rightarrow g_n\} : X \to Y + E$$

for each term $f : X \to Y + E$, each non-empty list of indices $(i_1, \dots, i_n)$ and each terms $g_j : P_{i_j} \to Y + E$ for $j = 1, \dots, n$ is defined in two steps:

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

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

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

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

Remark 2.10. When $n = 1$ we get simply:

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

which can be illustrated as follows, with $try\{f\}\,k$ on the left and $k = catch\,\{i \Rightarrow g\}$ on the right:

$$\begin{tikzcd}
& Y \arrow[d, "normal"'] \arrow[drr, "normal"] & & \\
X \arrow[r, "f"] & Y+E \arrow[rr, "{[normal|k]}"] \arrow[ur, phantom, "="] \arrow[dr, phantom, "="] & & Y+E \\
& E \arrow[u, "abrupt"] \arrow[urr, "k"'] & &
\end{tikzcd}
\qquad
\begin{tikzcd}
& P_i \arrow[d, "normal"'] \arrow[drr, "g"] & & \\
E \arrow[r, "c_i"] & P_i+E \arrow[rr, "{[g|abrupt]}"] \arrow[ur, phantom, "="] \arrow[dr, phantom, "="] & & Y+E \\
& E \arrow[u, "abrupt"] \arrow[urr, "abrupt"'] & &
\end{tikzcd}$$

Remark 2.11. About the handling function $try\{f\}\ catch\,\{i_1 \Rightarrow g_1 | \dots | i_n \Rightarrow g_n\}$, it should be noted that each g_i may itself raise exceptions and that the indices $i_1, \dots, i_n$ form a list: they are given in this order and they need not be pairwise distinct. It is assumed that this list is non-empty because it is the usual choice in programming languages, however it would be easy to drop this assumption.

2.5 The intended semantics of exceptions

As usual, a *Sig-algebra* M, for any signature *Sig*, is made of a set $M(X)$ for each type X in *Sig* and a function $M(f) : M(X_1) \times \cdots \times M(X_n) \to M(Y)$ for each operation $f : X_1, \ldots, X_n \to Y$. As in Definition 2.9, let Sig_{pure} be a signature and let Σ_{expl} be the explicit specification for exceptions associated to a family of pure types $(P_i)_{i \in I}$.

Definition 2.12. Given a Sig_{pure}-algebra M_{pure}, the *model of exceptions* M_{expl} of Σ_{expl} extending M_{pure} has its values in $\mathbf{Set}_{E,expl}$; it coincides with M_{pure} on Sig_{pure}, it interprets the type E as the disjoint union $E = \sum_{i \in I} P_i$ and the tagging operations $t_i : P_i \to E$ as the inclusions.

It follows that the interpretation of the tagging operation maps a non-exceptional value $a \in P_i$ to an exception $t_i(a) \in E$ (for clarity we keep the notation $t_i(a)$ instead of a). Then, because of the equations, the interpretation of the untagging operation $c_i : E \to P_i$ must proceed as follows: it checks whether its argument e is in the image of t_i, if this is the case then it returns the parameter $a \in P_i$ such that $e = t_i(a)$, otherwise it propagates the exception e. It is easy to check that the next Definition corresponds to the description of the mechanism of exceptions in Java: see remark 2.2 and Appendix A.

Definition 2.13. Given a signature Sig_{pure} and a Sig_{pure}-algebra M_{pure}, the *intended semantics of exceptions* is the model M_{expl} of the specification Σ_{expl} extending M_{pure}.

Remark 2.14. Let Sig_{exc} be the signature for exceptions as in Definition 2.1. It follows from Definition 2.13 that the intended semantics of exceptions cannot be seen as a Sig_{exc}-algebra. Indeed, although there is no type of exceptions in Sig_{exc}, the operation $throw_{Y,i} : P_i \to Y$ in Sig_{exc} has to be interpreted as a function $throw_{Y,i} : P_i \to Y + E$, where the set of exceptions E is usually non-empty.

2.6 About higher-order constructions

Definition 2.13 can easily be extended to a functional language. In order to add higher-order features to our explicit logic, let us introduce a functional type Z^W for each types W and Z. Then each $\varphi : W \to Z + E$ gives rise to $\lambda x.\varphi : \mathbb{1} \to (Z + E)^W$, which does not raise exceptions. It follows that $try\{\lambda x.\varphi\}\ catch\ \{i_1 \Rightarrow g_1 | \ldots | i_n \Rightarrow g_n\} \equiv \lambda x.\varphi$, which is the intended meaning of exceptions in functional languages like ML [11].

3 Exceptions as a computational effect

According to Definition 2.13, the intended semantics of exceptions can be defined in the explicit logic as a model M_{expl} of the explicit specification Σ_{expl}. However, by introducing a type of exceptions, the explicit logic does not take into account the fact that the exceptions form a computational effect: the model M_{expl} cannot be seen as an algebra of the signature Sig_{exc} for exceptions (Definition 2.1) since (denoting X for $M_{expl}(X)$ for each type X) the operation $throw_{Y,i} : P_i \to Y$ is interpreted as a function from P_i to $Y + E$ instead of from P_i to Y: this is a fundamental remark of Moggi in [15].

In this Section we build another logic $\mathcal{L}_{deco}$, called the *decorated* logic for exceptions, and a decorated specification Σ_{deco} for exceptions which reconciles the syntax and the semantics: Σ_{deco} fits with the syntax since it has no type of exceptions, and it provides the intended semantics because this semantics can be seen as a model M_{deco} of Σ_{deco}. In the decorated logic the terms and the equations are classified, or *decorated*, and their interpretation depends on their decoration.

The decorated logic is defined in Section 3.1. In Section 3.2 we define the decorated specification Σ_{deco} and the model M_{deco} of Σ_{deco} and we prove that M_{deco} provides the intended semantics of exceptions. Besides, we show in Section 3.4 that it is easy to relate the decorated specification Σ_{deco} to the signature for exceptions Sig_{exc}; for this purpose we introduce a logic $\mathcal{L}_{app}$, called the *apparent* logic, which is quite close to the monadic equational logic. This is illustrated by Figure 2, which extends Figure 1 by filling the gap between syntax and semantics. This is obtained by adding two morphisms of logic, $F_d : \mathcal{L}_{deco} \to \mathcal{L}_{app}$ on the syntax side and $F_e : \mathcal{L}_{deco} \to \mathcal{L}_{expl}$ on the semantics side. The rules of the decorated logic are used for proving some properties of exceptions in Section 3.5. The decorated logic is extended to higher-order features in Section 3.6.

$$\begin{tikzcd}
\textbf{syntax} & \textbf{syntax} \arrow[l, Leftarrow] & \\
 & \textbf{semantics} \arrow[r, Leftrightarrow] & \textbf{semantics} \\
\mathcal{L}_{app} & \mathcal{L}_{deco} \arrow[l, "F_d"'] \arrow[r, "F_e"] & \mathcal{L}_{expl} \\
Sig_{exc} \subseteq \Sigma_{app} & \Sigma_{deco} \arrow[l, mapsto, "F_d"'] \arrow[r, mapsto, "F_e"] \arrow[d, "M_{deco}"'] & \Sigma_{expl} \arrow[d, "M_{expl}"] \\
 & \Theta_{deco} & \Theta_{expl} \arrow[l, mapsto, "G_e"']
\end{tikzcd}$$

Figure 2: Syntax and semantics of exceptions, reconciled

3.1 Decorated logic for exceptions

Here we define the decorated logic for exceptions $\mathcal{L}_{deco}$, by giving its syntax and its inference rules, and we define a morphism from $\mathcal{L}_{deco}$ to $\mathcal{L}_{expl}$ for expliciting the meaning of the decorations. The syntax of $\mathcal{L}_{deco}$ consists in types, terms and equations, like $\mathcal{L}_{meq}$ in Example 1.6, but with three kinds of terms and two kinds of equations. The terms are decorated by (0), (1) and (2) used as superscripts, they are called respectively *pure* terms, *propagators* and *catchers*. The equations are denoted by two distinct relational symbols, $\equiv$ for *strong* equations and $\sim$ for *weak* equations.

The *expansion* functor is the locally presentable functor $F_{e,S} : \mathbf{S}_{deco} \to \mathbf{S}_{expl}$ defined in Figure 3 by mapping each elementary decorated specification (type, decorated term, decorated equation) to an explicit specification. Note: in the explicit specifications the type of exceptions E may be duplicated for readability, and the superscript (d) stands for any decoration. Thus, the expansion provides a meaning for the decorations:

(0) a *pure* term may neither raise exceptions nor recover form exceptions,

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

(2) a *catcher* may raise exceptions and recover form exceptions.

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

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

Remark 3.1. It happens that the image of a decorated term by the expansion morphism can be characterized by a term, so that we can say "for short" that the expansion of a catcher $f^{(2)} : X \to Y$ "is" $f : X+E \to Y+E$, the expansion of a propagator $f^{(1)} : X \to Y$ "is" $f_1 : X \to Y+E$ where $f_1 = f \circ normal_X$, and the expansion of a pure term $f^{(0)} : X \to Y$ "is" $f_0 : X \to Y$. In a similar way, we say that the expansion of a type Z "is" Z. This is stated in the last column of Figure 3. However this may lead to some misunderstanding. Indeed, while the image of a specification by the expansion morphism must be a specification, the image of a type does not have to be a type and the image of a term does not have to be a term.

The rules of $\mathcal{L}_{deco}$ are given in Figure 4. 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. According to Definition 1.8, the expansion morphism maps each inference rule of $\mathcal{L}_{expl}$ to a proof in $\mathcal{L}_{expl}$; this provides the meaning of the decorated rules:

(a) The first part of the decorated monadic equational rules for exceptions are the rules for the monadic equational logic; this means that the catchers satisfy the monadic equational rules with respect to the strong equations.

	Σ_{deco}	$F_{e,S}\Sigma_{deco}$	$F_{e,S}\Sigma_{deco}$ "for short"
type	Z	$\begin{tikzcd} Z \arrow[d, "normal"] \\ Z+E \\ E \arrow[u, "abrupt"] \end{tikzcd}$	Z
catcher	$X \xrightarrow{f^{(2)}} Y$	$\begin{tikzcd} X \arrow[d] & Y \arrow[d] \\ X+E \arrow[r, "f"] & Y+E \\ E \arrow[u] & E \arrow[u] \end{tikzcd}$	$X+E \xrightarrow{f} Y+E$
propagator	$X \xrightarrow{f^{(1)}} Y$	$\begin{tikzcd} X \arrow[d] & Y \arrow[d] \\ X+E \arrow[r, "f"] \arrow[dr, phantom, "\equiv"] & Y+E \\ E \arrow[u] \arrow[r, "id"'] & E \arrow[u] \end{tikzcd}$	$X \xrightarrow{f_1 = f \circ normal} Y+E$
pure term	$X \xrightarrow{f^{(0)}} Y$	$\begin{tikzcd} X \arrow[d] \arrow[r, "f_0"] \arrow[dr, phantom, "\equiv"] & Y \arrow[d] \\ X+E \arrow[r, "f"] \arrow[dr, phantom, "\equiv"] & Y+E \\ E \arrow[u] \arrow[r, "id"'] & E \arrow[u] \end{tikzcd}$	$X \xrightarrow{f_0} Y$
strong equation	$f^{(d)} \equiv g^{(d)} :$ $X \to Y$	$f \equiv g :$ $X+E \to Y+E$	$f \equiv g$
weak equation	$f^{(d)} \sim g^{(d)} :$ $X \to Y$	$f \circ normal_X \equiv g \circ normal_X :$ $X \to Y+E$	$f_1 \equiv g_1$

Figure 3: The expansion morphism

(b) The second part of the decorated monadic equational rules for exceptions deal with the conversions between decorations and with the equational-like properties of pure operations, propagators and 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.

(c) The rules for the propagation of exceptions build a propagator $\blacktriangledown k$ from any catcher k. The expansion of $\blacktriangledown k$ is defined as $[k \circ normal_X | abrupt_X] : X+E \to Y+E$: it coincides with the expansion of k on X and it propagates exceptions without catching them, otherwise.

(d) The rules for a decorated initial type $\mathbb{0}$ together with the rules in (b) imply that every propagator from $\mathbb{0}$ to any X is strongly equivalent to $[\,]_X$. The expansion of $\mathbb{0}$ and $[\,]_X^{(0)}$ are the initial type $\mathbb{0}$ and the term $[\,]_X$, respectively, as in remark 2.8.

(e) The pure coproduct $(id_X : X \to X + \mathbb{0} \leftarrow \mathbb{0} : [\,]_X)$ has decorated coproduct properties which are given by the rules for the case distinction with respect to $X + \mathbb{0}$. The expansion of $[g|k]^{(2)} : X \to Y$ is the case distinction $[g_1|k] : X+E \to Y+E$ with respect to $X+E$ (where $\mathbb{0}+E$ is identified with E, so that $k : E \to Y+E$). This can be illustrated as follows, by a diagram in the decorated logic (on the

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

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

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

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

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

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

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

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

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

(c) Rules for the propagation of exceptions

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

(d) Rules for a decorated initial type $\mathbb{0}$

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

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

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

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

(f) Rules for a constitutive coproduct $(q_i^{(1)} : X_i \to X)_i$

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

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

Figure 4: Decorated rules for exceptions

left) or in the explicit logic (on the right); more details are given in Remark 3.2.

$$\begin{tikzcd}
X \arrow[d, "id^{(0)}"'] \arrow[drr, "g^{(1)}"] & \sim & & & X \arrow[d, "normal"'] \arrow[drr, "g_1"] & \equiv & \\
X \arrow[rr, "{[g|k]^{(2)}}"] & & Y & & X+E \arrow[rr, "{[g_1|k]}"] & & Y+E \\
\mathbb{0} \arrow[u, "{[\,]^{(0)}}"] \arrow[urr, "k^{(2)}"'] & \equiv & & & E \arrow[u, "abrupt"] \arrow[urr, "k"'] & \equiv &
\end{tikzcd} \tag{1}$$

(f) The rules for a constitutive coproduct build a catcher from a family of propagators. Whenever $(q_i^{(1)} : X_i \to X)_i$ is a constitutive coproduct the family $(q_{i,1} : X_i \to X+E)_i$ is a coproduct with respect to the explicit logic.

Remark 3.2. Let us give some additional information on the expansion of the decorated rules (e) in Figure 4, i.e., the decorated rules for case distinction with respect to $X+\mathbb{0}$. According to the definition of the expansion morphism on specifications (Figure 3) since the cocone $(id_X^{(0)} : X \to X+\mathbb{0} \leftarrow \mathbb{0} : [\,]_X^{(0)})$ is made of pure terms, we can say "for short" that its expansion "is" simply $(id_{X,0} : X \to X+\mathbb{0} \leftarrow \mathbb{0} : [\,]_{X,0})$. However in order to check that the decorated rules (e) in Figure 4 are mapped by the expansion morphism to explicit proofs we have to take into account another coproduct in the explicit logic. Rules (e) state that *for each propagator $g^{(1)} : X \to Y$ and each catcher $k^{(2)} : \mathbb{0} \to Y$ there is a catcher $h^{(2)} : X \to Y$ (h is denoted $[g|k]$ in Figure 4) such that $h \sim g$ and $h \circ [\,]_X \equiv k$, and that in addition h is, up to strong equivalence, the unique catcher satisfying these conditions.* Thus, according to Figure 3, the expansion of these rules must be such that *for each terms $g_1 : X \to Y+E$ and $k : E \to Y+E$ there is a term $h : X+E \to Y+E$ such that $h \circ normal_X \equiv g \circ normal_X$ and $h \circ abrupt_X \equiv k$, and that in addition h is, up to equivalence, the unique term satisfying these conditions.* Clearly, this is satisfied when $h = [g_1|h]$ is obtained by case distinction with respect to the coproduct $(normal_X : X \to X+E \leftarrow E : abrupt_X)$. It follows that we can also say, "for short", that the image of the coproduct $(id_X : X \to X+\mathbb{0} \leftarrow \mathbb{0} : [\,]_X)$ by the expansion morphism "is" the coproduct $(normal_X : X \to X+E \leftarrow E : abrupt_X)$, as in diagram (1).

The decorated rules are now used for proving a lemma that will be used in Section 3.2.

Lemma 3.3. *For each propagator $g^{(1)} : X \to Y$ we have $g \circ [\,]_X \equiv [\,]_Y$ and $g \equiv [g \mid [\,]_Y]$.*

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

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

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

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

□

Remark 3.4. The morphism of limit sketches $\mathbf{e} : \mathbf{E}_S \to \mathbf{E}_T$ which induces the decorated logic is easily guessed. This is outlined below, more details are given in a similar exercice in [2]. The description of $\mathbf{E}_S$ can be read from the second column of Figure 3. There is in the limit sketch $\mathbf{E}_S$ a point for each elementary decorated specification and an arrow for each morphism between the elementary specifications, in a contravariant way. For instance $\mathbf{E}_S$ has points *type* and *catcher*, and it has arrows *source* and *target* from *catcher* to *type*, corresponding to the morphisms from the decorated specification Z to the decorated specification $f^{(2)} : X \to Y$ which map Z respectively to X and Y. As usual, some additional points, arrows and distinguished cones are required in $\mathbf{E}_S$. The description of $\mathbf{e}$ can be read from Figure 4. The morphism $\mathbf{e}$ adds inverses to arrows in $\mathbf{E}_S$ corresponding to the inference rules, in a way similar to Example 1.6 but in a contravariant way.

Remark 3.5. In the short note [3] it is checked that, from a denotational point of view, the functions for tagging and untagging exceptions are respectively *dual*, in the categorical sense, to the functions for looking up and updating states. This duality relies on the fact that the states are *observed* thanks to the lookup operations while dually the exceptions are *constructed* thanks to the tagging operations. Thus, the duality between states and exceptions stems from the duality between the comonad $X \times S$ (for some fixed S) and the monad $X + E$ (for some fixed E). It happens that this duality also holds from the decorated point of view.

Most of the decorated rules for exceptions are dual to the decorated rules for states in [4]. For instance, the unique difference between the monadic equational rules for exceptions (parts (a) and (b) of Figure 4) and the dual rules for states in [4] lies in the congruence rules for the weak equations: for states the replacement rule is restricted to pure g, while for exceptions it is the substitution rule which is restricted to pure f. The rules for a decorated initial type and for a constitutive coproduct (parts (d) and (f) of Figure 4) are respectively dual to the rules for a decorated final type and the rules for an observational product in [4]. The rules for the propagation of exceptions and for the case distinction with respect to $X + \mathbb{0}$ (parts (c) and (e) of Figure 4) are used only for the construction of the handling operations from the untagging operations; these rules have no dual in [4] for states.

Remark 3.6. For a while, let us forget about the three last families of rules in Figure 4, which involve some kind of decorated coproduct. Then any monad T on any category $\mathbf{C}$ provides a decorated theory $\mathbf{C}_T$, as follows. The types are the objects of $\mathbf{C}$, a pure term $f^{(0)} : X \to Y$ is a morphism $f : X \to Y$ in $\mathbf{C}$, a propagator $f^{(1)} : X \to Y$ is a morphism $f : X \to TY$ in $\mathbf{C}$, a catcher $f^{(2)} : X \to Y$ is a morphism $f : TX \to TY$ in $\mathbf{C}$. The conversion from pure to propagator uses the unit of T and the conversion from propagator to catcher uses the multiplication of T. Composition of propagators is done in the Kleisli way. A strong equation $f^{(2)} \equiv g^{(2)} : X \to Y$ is an equality $f = g : TX \to TY$ in $\mathbf{C}$ and a weak equation $f^{(2)} \sim g^{(2)} : X \to Y$ is an equality $f \circ \eta_X = g \circ \eta_X : X \to TY$ in $\mathbf{C}$, where η is the unit of the monad. It is easy to check that the decorated monadic equational rules of $\mathcal{L}_{deco}$ are satisfied, as well as the rules for the propagation of exceptions if $\blacktriangledown k = k \circ \eta_X : X \to TY$ for each $k : TX \to TY$.

3.2 Decorated specification for exceptions

Let us define a decorated specification Σ_{deco} for exceptions, which (like Σ_{expl} in Section 2.4) defines the raising and handling operations in terms of the core tagging and untagging operations.

Definition 3.7. Let Sig_{pure} be a signature. Given a set of indices I and a type P_i in Sig_{pure} for each $i \in I$, the *decorated specification for exceptions* Σ_{deco} is the $\mathcal{L}_{deco}$-specification made of Sig_{pure} with its operations decorated as pure together with, for each $i \in I$, a propagator $t_i^{(1)} : P_i \to \mathbb{0}$ and a catcher $c_i^{(2)} : \mathbb{0} \to P_i$ with the weak equations $c_i \circ t_i \sim id : P_i \to P_i$ and $c_i \circ t_j \sim [\,] \circ t_j : P_j \to P_i$ for all $j \neq i$. Then for each $i \in I$ the *raising* propagator $(throw_{Y,i})^{(1)} : P_i \to Y$ for each type Y in Sig_{pure} is:

$$throw_{Y,i} = [\,]_Y \circ t_i$$

and the *handling* propagator $(try\{f\}\ catch\,\{i_1 \Rightarrow g_1 | \dots | i_n \Rightarrow g_n\})^{(1)} : X \to Y$ for each propagator $f^{(1)} : X \to Y$, each non-empty list of indices $(i_1, \dots, i_n)$ and each propagators $g_j^{(1)} : P_{i_j} \to Y$ for $j = 1, \dots, n$ is defined as:

$$try\{f\}\ catch\,\{i_1 \Rightarrow g_1|\dots|i_n \Rightarrow g_n\} = \blacktriangledown TRY\{f\}\ catch\,\{i_1 \Rightarrow g_1|\dots|i_n \Rightarrow g_n\}$$

from a catcher $TRY\{f\}\ catch\,\{i_1 \Rightarrow g_1|\dots|i_n \Rightarrow g_n\} : X \to Y$ which is defined as follows in two steps:

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

$$(TRY\{f\}\,k)^{(2)} \;=\; \left[\ id_Y^{(0)} \mid k^{(2)}\ \right]^{(2)} \circ f^{(1)}$$

(catch) the catcher $catch\,\{i_1 \Rightarrow g_1|\dots|i_n \Rightarrow g_n\} : \mathbb{0} \to Y$ is obtained by setting $p = 1$ in the family of catchers $k_p = catch\,\{i_p \Rightarrow g_p|\dots|i_n \Rightarrow g_n\} : \mathbb{0} \to Y$ (for $p = 1, \dots, n+1$) which are defined recursively by:

$$k_p^{(2)} \;=\; \begin{cases} [\,]_Y^{(0)} & \text{when } p = n+1 \\ \left[\ g_p^{(1)} \mid k_{p+1}^{(2)}\ \right]^{(2)} \circ c_{i_p}^{(2)} & \text{when } p \le n \end{cases}$$

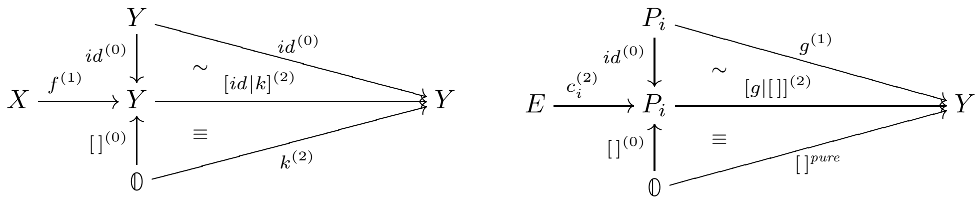
Remark 3.8. Let $h = try\{f\}\ catch\,\{i_1 \Rightarrow g_1|\dots|i_n \Rightarrow g_n\}$ and $H = TRY\{f\}\ catch\,\{i_1 \Rightarrow g_1|\dots|i_n \Rightarrow g_n\}$. Then h is a propagator and H is a catcher, and the definition of h is given in terms of H, as $h = \blacktriangledown H$. The expansions of h and H are functions from $X + E$ to $Y + E$ which coincide on X but differ on E: while h propagates exceptions, H catches exceptions according to the pattern $catch\,\{i_1 \Rightarrow g_1|\dots|i_n \Rightarrow g_n\}$.

Remark 3.9. Since $k_{n+1} = [\,]_Y$, by Lemma 3.3 we have $[g_n|k_{n+1}] \equiv g_n$. It follows that when $n = 1$ and 2 we get respectively:

$$try\{f\}\ catch\,\{i \Rightarrow g\} \;\equiv\; \blacktriangledown\big(\ [id_Y \mid g \circ c_i] \circ f\ \big) \tag{2}$$

$$try\{f\}\ catch\,\{i \Rightarrow g \mid j \Rightarrow h\} \;\equiv\; \blacktriangledown\big(\ [id \mid [g \mid h \circ c_j] \circ c_i] \circ f\ \big) \tag{3}$$

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



$$\begin{tikzcd} & Y \arrow[d, "id^{(0)}"'] \arrow[drr, "id^{(0)}"] & \sim & \\ X \arrow[r, "f^{(1)}"] & Y \arrow[rr, "{[id|k]^{(2)}}"] & & Y \\ & \mathbb{0} \arrow[u, "{[\,]^{(0)}}"] \arrow[urr, "k^{(2)}"'] & \equiv & \end{tikzcd} \qquad \begin{tikzcd} & P_i \arrow[d, "id^{(0)}"'] \arrow[drr, "g^{(1)}"] & \sim & \\ E \arrow[r, "c_i^{(2)}"] & P_i \arrow[rr, "{[g|[\,]]^{(2)}}"] & & Y \\ & \mathbb{0} \arrow[u, "{[\,]^{(0)}}"] \arrow[urr, "{[\,]^{pure}}"'] & \equiv & \end{tikzcd}$$

Lemma 3.10. *Let Sig_{pure} be a signature, I a set and P_i a type in Sig_{pure} for each $i \in I$. Let Σ_{expl} be the corresponding explicit specification for exceptions (Definition 2.9) and Σ_{deco} the corresponding decorated specification for exceptions (Definition 3.7). Then $\Sigma_{expl} = F_e \Sigma_{deco}$.*

Proof. This is easy to check: in Definition 2.9 Σ_{expl} is described as a colimit of elementary specifications, and F_e, as any left adjoint functor, preserves colimits. □

Proposition 3.11. *The functor $F_{e,S} : \mathbf{S}_{deco} \to \mathbf{S}_{expl}$ defined in Figure 3 is locally presentable and it determines a morphism of logics $F_e : \mathcal{L}_{deco} \to \mathcal{L}_{expl}$.*

Proof. The fact that $F_{e,S}$ is locally presentable is deduced from its definition in Figure 3. It has been checked that $F_{e,S}$ maps each decorated inference rule to an explicit proof, thus it can be extended as $F_{e,T} : \mathbf{T}_{deco} \to \mathbf{T}_{expl}$ in such a way that the pair $F_e = (F_{e,S}, F_{e,T})$ is a morphism of logics. □

Definition 3.12. The morphism $F_e : \mathcal{L}_{deco} \to \mathcal{L}_{expl}$ is called the *expansion* morphism.

3.3 The decorated model provides the intented semantics of exceptions

Following Definition 2.13, the intended semantics of exceptions is a model with respect to the explicit logic. Theorem 3.15 will prove that the intended semantics of exceptions can also be expressed as a model with respect to the decorated logic.

Definition 3.13. For any set E, called the *set of exceptions*, we define a decorated theory $\mathbf{Set}_{E,deco}$ as follows. A type is a set, a pure term $f^{(0)} : X \to Y$ is a function $f : X \to Y$, a propapagator $f^{(1)} : X \to Y$ is a function $f : X \to Y + E$, and a catcher $f^{(2)} : X \to Y$ is a function $f : X + E \to Y + E$. It follows that in $\mathbf{Set}_{E,deco}$ every pure term $f : X \to Y$ gives rise to a propagator $normal_Y \circ f : X \to Y + E$ and that every propagator $f : X \to Y + E$ gives rise to a catcher $[f|abrupt_Y] : X + E \to Y + E$. By default, f stands for $f^{(2)}$. The equations are defined when both members are catchers, the other cases follow thanks to the conversions above. A strong equation $f \equiv g : X \to Y$ is the equality of functions $f = g : X + E \to Y + E$ and a weak equation $f \sim g : X \to Y$ is the equality of functions $f \circ normal_X = g \circ normal_X : X \to Y + E$.

Lemma 3.14. *Let $G_{e,T}$ be the right adjoint to $F_{e,T}$. Then $\mathbf{Set}_{E,deco} = G_{e,T}\mathbf{Set}_{E,expl}$.*

Proof. The morphism of limit sketches φ_e, corresponding to the locally presentable functor $F_{e,T}$, is deduced from Figure 3. By definition of $G_{e,T}$ we have $G_{e,T}\mathbf{Set}_{E,expl} = \mathbf{Set}_{E,expl} \circ \varphi_e$. The lemma follows by checking that the definition of $\mathbf{Set}_{E,deco}$ (Definition 3.13) is precisely the description of $\mathbf{Set}_{E,expl} \circ \varphi_e$. □

Our main result is the next theorem, which states that the decorated point of view provides exactly the semantics of exceptions defined as a model of the explicit specification for exceptions in Definition 2.13. Thus the decorated point of view *is* an alternative to the explicit point of view, as it provides the intended semantics, but it is also closer to the syntax since the type of exceptions is no longer explicit.

To prove this, the key point is the existence of the expansion morphism from the decorated to the explicit logic. Within the category of diagrammatic logics, the proof is simple: it uses the fact that the expansion morphism, like every morphism in this category, is a left adjoint functor.

Theorem 3.15. *The model M_{deco} of the specification Σ_{deco} with values in the theory $\mathbf{Set}_{E,deco}$ in the decorated logic provides the intended semantics of exceptions.*

Proof. According to Definition 2.13, the intended semantics of exceptions is the model M_{expl} of Σ_{expl} with values in $\mathbf{Set}_{E,expl}$ in the explicit logic. In addition, M_{deco} is a model of Σ_{deco} with values in $\mathbf{Set}_{E,deco}$ in the decorated logic. Furthermore, we know from Lemmas 3.10 and 3.14 that $\Sigma_{expl} = F_e\Sigma_{deco}$ and $\mathbf{Set}_{E,deco} = G_e\mathbf{Set}_{E,expl}$, where G_e is right adjoint to F_e. Thus, it follows from proposition 1.9 that there is a bijection between $Mod_{\mathcal{L}_{expl}}(\Sigma_{expl}, \mathbf{Set}_{E,expl})$ and $Mod_{\mathcal{L}_{deco}}(\Sigma_{deco}, \mathbf{Set}_{E,deco})$. Finally, it is easy to check that M_{deco} corresponds to M_{expl} in this bijection. □

3.4 The decorated syntax provides the syntax of exceptions

The signature Sig_{exc} from Definition 2.1 can easily be recovered from the decorated specification Σ_{deco} by dropping the decorations and forgetting the equations. More formally, this can be stated as follows. Let us introduce a third logic $\mathcal{L}_{app}$, called the *apparent* logic, by dropping all the decorations from the decorated logic; thus, the apparent logic is essentially the monadic equational logic with an empty type. The fact of dropping the decorations is a morphism of logics $F_d : \mathcal{L}_{deco} \to \mathcal{L}_{app}$. Therefore, we can form the apparent specification $\Sigma_{app} = F_d\Sigma_{deco}$ which contains the signature for exceptions Sig_{exc}. Note that, as already mentioned in Remark 2.14, the intended semantics of exceptions cannot be seen as a set-valued model of Σ_{app}.

3.5 Some decorated proofs for exceptions

According to Theorem 3.15, the intended semantics of exceptions can be expressed as a model in the decorated logic. Now we show that the decorated logic can also be used for proving properties of exceptions in a concise way. Indeed, as for proofs on states in [4], we may consider two kinds of proofs on exceptions: the *explicit* proofs involve a type of exceptions, while the *decorated* proofs do not mention any type of exceptions but require the specification to be decorated, in the sense of Section 3. In addition, the expansion morphism, from the decorated logic to the explicit logic, maps each decorated proof to an explicit one. In this Section we give some decorated proofs for exceptions, using the inference rules of Section 3.1.

We know from [3] that the properties of the core tagging and untagging operations for exceptions are dual to the properties of the lookup and update operations for states. Thus, we may reuse the decorated proofs involving states from [4]. Starting from any one of the seven equations for states in [16] we can dualize this equation and derive a property about raising and handling exceptions. This is done here for the *annihilation catch-raise* and for the *commutation catch-catch* properties.

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 [4]. By duality we get the following *annihilation untag-tag* property (Lemma 3.16), which means that tagging just after untagging, both with respect to the same index, returns the given exception. Then this result is used in Proposition 3.17 for proving the *annihilation catch-raise* property: catching an exception and re-raising it is like doing nothing.

Lemma 3.16 (Annihilation untag-tag)**.** *For each $i \in I$:*

$$t_i^{(1)} \circ c_i^{(2)} \equiv id_{\mathbb{0}}^{(0)} \ .$$

Proposition 3.17 (Annihilation catch-raise)**.** *For each propagator $f^{(1)} : X \to Y$ and each $i \in I$:*

$$\mathit{try}\{f\}\ \mathit{catch}\,\{i \Rightarrow \mathit{throw}_{Y,i}\} \equiv f \ .$$

Proof. By Equation (2) and Definition 3.7 we have $\mathit{try}\{f\}\ \mathit{catch}\,\{i \Rightarrow \mathit{throw}_{Y,i}\} \equiv \blacktriangledown([id_Y|\,[\,]_Y \circ t_i \circ c_i] \circ f)$. By Lemma 3.16 $[id_Y|\,[\,]_Y \circ t_i \circ c_i] \equiv [id_Y|\,[\,]_Y]$, and the unicity property of $[id_Y|\,[\,]_Y]$ implies that $[id_Y|\,[\,]_Y] \equiv id_Y$. Thus $\mathit{try}\{f\}\ \mathit{catch}\,\{i \Rightarrow \mathit{throw}_{Y,i}\} \equiv \blacktriangledown f$. In addition, since $\blacktriangledown f \sim f$ and f is a propagator we get $\blacktriangledown f \equiv f$. Finally, the transitivity of $\equiv$ yields the proposition. □

On states, the *commutation update-update* property means that updating two different locations can be done in any order. By duality we get the following *commutation untag-untag* property, (Lemma 3.18) which means that untagging with respect to two distinct exceptional types can be done in any order. A detailed decorated proof of the commutation update-update property is given in [4]. The statement of this property and its proof use *semi-pure products*, which were introduced in [5] 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 coproduct $X + \mathbb{0}$.

The *coproduct* of two types A and B is defined as a type $A{+}B$ with two pure coprojections $q_1^{(0)} : A \to A{+}B$ and $q_2^{(0)} : B \to A + B$, which satisfy the usual categorical coproduct property with respect to the pure morphisms. Then the *semi-pure coproduct* of a propagator $f^{(1)} : A \to C$ and a catcher $k^{(2)} : B \to C$ is a catcher $[f|k]^{(2)} : A + B \to C$ which is characterized, up to strong equations, by the following decorated version of the coproduct property: $[f|k] \circ q_1 \sim f$ and $[f|k] \circ q_2 \equiv k$. Then as usual, the coproduct $f' + k' : A{+}B \to C{+}D$ of a propagator $f' : A \to C$ and a catcher $k' : B \to D$ is the catcher $f' + k' = [q_1 \circ f \mid q_2 \circ k] : A + B \to C + D$.

Whenever f and g are propagators it can be proved that $\blacktriangledown\,[f|g] \equiv [f|g]$; 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 3.18 (Commutation untag-untag)**.** *For each $i, j \in I$ with $i \neq j$:*

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

Proposition 3.19 (Commutation catch-catch)**.** *For each $i, j{\in}I$ with $i{\neq}j$:*

$$\mathit{try}\{f\}\ \mathit{catch}\,\{i{\Rightarrow}g \mid j{\Rightarrow}h\} \equiv \mathit{try}\{f\}\ \mathit{catch}\,\{j{\Rightarrow}h \mid i{\Rightarrow}g\}$$

Proof. According to Equation (3): $try\{f\}\, catch\, \{i \Rightarrow g \mid j \Rightarrow h\} \equiv \blacktriangledown([id \mid [g \mid h \circ c_j] \circ c_i] \circ f)$. Thus, the result will follow from $[g \mid h \circ c_j] \circ c_i \equiv [h \mid g \circ c_i] \circ c_j$. It is easy to check that $[g \mid h \circ c_j] \equiv [g \mid h] \circ (id_{P_i} + c_j)$, so that $[g \mid h \circ c_j] \circ c_i \equiv [g \mid h] \circ (id_{P_i} + c_j) \circ c_i$. Similarly $[h \mid g \circ c_i] \circ c_j \equiv [h \mid g] \circ (id_{P_j} + c_i) \circ c_j$ hence $[h \mid g \circ c_i] \circ c_j \equiv [g \mid h] \circ (c_i + id_{P_j}) \circ c_j$. Then the result follows from Lemma 3.18. □

3.6 About higher-order constructions

We know from Section 2.6 that we can add higher-order features in our explicit logic. This remark 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 expansion of $Z^{W(0)}$ is Z^W, the expansion of $Z^{W(1)}$ is $(Z+E)^W$ and the expansion of $Z^{W(2)}$ is $(Z+E)^{(W+E)}$. Then each $\varphi^{(d)} : W \to Z$ gives rise to $\lambda x.\varphi : \mathbb{1} \to Z^{W(d)}$, and a major point is that $\lambda x.\varphi$ is pure for every decoration (d) of φ. Informally, we can say that the abstraction moves the decoration from the term to the type. This means that the expansion of $(\lambda x.\varphi)^{(0)}$ is $\lambda x.\varphi : \mathbb{1} \to F_e(Z^{W(d)})$, as required: for instance when $\varphi^{(1)}$ is a propagator the expansion of $(\lambda x.\varphi)^{(0)}$ is $\lambda x.\varphi : \mathbb{1} \to (Z+E)^W$, as in Section 2.6. Besides, it is easy to prove in the decorated logic that whenever f is pure we get $try\{f\}\, catch\, \{i_1 \Rightarrow g_1 | \dots | i_n \Rightarrow g_n\} \equiv f$. It follows that this occurs when f is a lambda abstraction: $try\{\lambda x.\varphi\}\, catch\, \{i_1 \Rightarrow g_1 | \dots | i_n \Rightarrow g_n\} \equiv \lambda x.\varphi$, as expected in functional languages.

4 Conclusion and future work

We have presented three logics for dealing with exceptions: the *apparent* logic $\mathcal{L}_{app}$ (Section 3.4) for dealing with the syntax, the *explicit* logic $\mathcal{L}_{expl}$ (Section 2.3) for providing the semantics of exceptions as a model in a transparent way, and the *decorated* logic $\mathcal{L}_{deco}$ (Section 3.1) for reconciling syntax and semantics. These logics are related by morphisms of logics $F_d : \mathcal{L}_{deco} \to \mathcal{L}_{app}$ and $F_e : \mathcal{L}_{deco} \to \mathcal{L}_{expl}$. A similar approach can be used for other exceptions [1, 4].

Future work include the following topics.

- Dealing with n-ary operations involving exceptions. We can add a cartesian structure to our decorated logic thanks to the notion of *sequential product* from [5]. This notion is based on the *semi-pure products*, which are dual to the semi-pure coproducts used in Section 3.5.
- Adding higher-order features. This has been outlined in Sections 2.6 and 3.6, however a more precise comparison with [19] remains to be done.
- Deriving a decorated operational semantics for exceptions by directing the weak and strong equations.
- Using a proof assistant for decorated proofs. Thanks to the morphism $F_d : \mathcal{L}_{deco} \to \mathcal{L}_{app}$, checking a decorated proof can be split in two parts: first checking the undecorated proof in the apparent logic, second checking that the decorations can be added. This separation simplifies the definition of the formalization towards a proof assistant: first formalize the syntactic rules of the language, second add computational effects.
- Combining computational effects. Since an effect is based on a span of logics, the combination of effects might be based on the composition of spans.

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

References

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

[2] 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).

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

[7] Charles Ehresmann. Esquisses et types de structures algébriques. Bull. Instit. Polit. Iaşi XIV (1968).

[8] P. Gabriel and F. Ulmer. Lokal präsentierbar Kategorien. Springer Lecture Notes in Mathematics 221 (1971).

[9] Peter Gabriel, Michel Zisman. Calculus of Fractions and Homotopy Theory. Springer (1967).

[10] 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`.

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

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

[13] 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).

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

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

[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] Jon G. Riecke, Hayo Thielecke. Typed Exceptions and Continuations Cannot Macro-Express Each Other. ICALP 1999 Springer-Verlag Lecture Notes in Computer Science 1644, p. 635-644 (1999).

[20] 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).

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

[22] Charles Wells. Sketches: Outline with references. `http://www.cwru.edu/artsci/math/wells/pub/papers.html` (1994).

A Handling exceptions in Java

Definition 2.13 relies on the following description of the handling of exceptions in Java [10, Ch. 14].

> A try statement without a finally block is executed by first executing the try block. Then there is a choice:
>
> 1. If execution of the try block completes normally, then no further action is taken and the try statement completes normally.
> 2. If execution of the try block completes abruptly because of a throw of a value V, then there is a choice:
>   (a) If the run-time type of V is assignable to the parameter of any catch clause of the try statement, then the first (leftmost) such catch clause is selected. The value V is assigned to the parameter of the selected catch clause, and the block of that catch clause is executed.
>   i. If that block completes normally, then the try statement completes normally;
>   ii. if that block completes abruptly for any reason, then the try statement completes abruptly for the same reason.
>   (b) If the run-time type of V is not assignable to the parameter of any catch clause of the try statement, then the try statement completes abruptly because of a throw of the value V.
> 3. If execution of the try block completes abruptly for any other reason, then the try statement completes abruptly for the same reason.

In fact, points 2(a)i and 2(a)ii can be merged. Our treatment of exceptions is similar to the one in Java when execution of the try block completes normally (point 1) or completes abruptly because of a throw of an exception of constructor $i \in I$ (point 2): indeed, in our framework there is no other reason for the execution of a try block to complete abruptly (point 3). Thus, the description can be simplified as follows.

> A try statement without a finally block is executed by first executing the try block. Then there is a choice:
>
> 1. If execution of the try block completes normally, then no further action is taken and the try statement completes normally.
> 2. If execution of the try block completes abruptly because of a throw of a value V, then there is a choice:
>   (a) If the run-time type of V is assignable to the parameter of any catch clause of the try statement, then the first (leftmost) such catch clause is selected. The value V is assigned to the parameter of the selected catch clause, the block of that catch clause is executed, and the try statement completes in the same way as this block.
>   (b) If the run-time type of V is not assignable to the parameter of any catch clause of the try statement, then the try statement completes abruptly because of a throw of the value V.

This simplified description corresponds to the definition of $\mathit{try}\{f\}\,\mathit{catch}\,\{i_1 \Rightarrow g_1|\dots|i_n \Rightarrow g_n\}$ in Definition 2.9, with points 1 and 2 corresponding respectively to **(try)** and **(catch)**.