States and exceptions considered as dual effects

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

States and exceptions considered as dual effects

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

May 19., 2011

Abstract

Abstract. In this paper we consider the two major computational effects of states and exceptions, from the point of view of diagrammatic logics. We get a surprising result: there exists a symmetry between these two effects, based on the well-known categorical duality between products and coproducts. More precisely, the lookup and update operations for states are respectively dual to the throw and catch operations for exceptions. This symmetry is deeply hidden in the programming languages; in order to unveil it, we start from the monoidal equational logic and we add progressively the logical features which are necessary for dealing with either effect. This approach gives rise to a new point of view on states and exceptions, which bypasses the problems due to the non-algebraicity of handling exceptions.

Introduction

In this paper we consider two major computational effects: states and exceptions. We get a surprising result: there exists a symmetry between these two effects, based on the well-known categorical duality between products and coproducts (or sums).

In order to get these results we use the categorical approach of diagrammatic logics, as introduced in [Duval 2003] and developed in [Domínguez & Duval 2010]. For instance, in [Dumas et al. 2011] this approach is used for studying an issue related to computational effects: controling the order of evaluation of the arguments of a function. This paper provides one more application of diagrammatic logics to computational effects; a preliminary approach can be found in [Duval & Reynaud 2005].

To our knowledge, the first categorical treatment of computational effects is due to Moggi [Moggi 1989, Moggi 1991]; this approach relies on monads, it is implemented in the programming language Haskell [Wadler 1992, Haskell]. Although monads are not used in this paper, the basic ideas underlying our approach rely on Moggi's remarks about notions of computations and monads. In view of comparing Moggi's approach and ours, let us quote [Moggi 1991, section 1]. *The basic idea behind the categorical semantics below is that, in order to interpret a programming language in a category* $\mathbf{C}$*, we distinguish the object* A *of values (of type* A*) from the object* TA *of computations (of type* A*), and take as denotations of programs (of type* A*) the elements of* TA*. In particular, we identify the type* A *with the object of values (of type* A*) and obtain the object of computations (of type* A*) by applying an unary type-constructor* T *to* A*. We call* T *a notion of computation, since it abstracts away from the type of values computations may produce. There are many choices for* TA *corresponding to different notions of computations. [...] Since the denotation of programs of type* B *are supposed to be elements of* TB*, programs of type* B *with a parameter of type* A *ought to be interpreted by morphisms with codomain* TB*, but for their domain there are two alternatives, either* A *or* TA*, depending on whether parameters of type* A *are identified with values or computations of type* A*.*

[*]LJK, Université de Grenoble, France. Jean-Guillaume.Dumas@imag.fr
[†]LJK, Université de Grenoble, France. Dominique.Duval@imag.fr
[‡]LJK, Université de Grenoble, France. Laurent.Fousse@imag.fr
[§]Malhivert, Claix, France. Jean-Claude.Reynaud@imag.fr

We choose the first alternative, because it entails the second. Indeed computations of type A are the same as values of type TA. The examples proposed by Moggi include the side-effects monad $TA = (A \times S)^S$ where S is the set of states and the exceptions monad $TA = A + E$ where E is the set of exceptions.

Later on, using the correspondence between monads and algebraic theories, Plotkin and Power proposed to use Lawvere theories for dealing with the operations and equations related to computational effects [Plotkin & Power 2002, Hyland & Power 2007]. The operations *lookup* and *update* are related to states, and the operations *raise* and *handle* are related to exceptions. In this framework, an operation is called *algebraic* when it satisfies some relevant genericity properties. It happens that *lookup*, *update* and *raise* are algebraic, while *handle* is not [Plotkin & Power 2003]. It follows that the handling of exceptions is quite difficult to formalize in this framework; several solutions are proposed in [Schröder & Mossakowski 2004, Levy 2006, Plotkin & Pretnar 2009]. In these papers, the duality between states and exceptions does not show up. One reason might be that, as we will see in this paper, exceptions catching is encapsulated in several nested conditionals which hide this duality.

Let us look more closely at the monad of exceptions $TA = A + E$. According to the point of view of monads for effects, a morphism from A to TB provides a denotation for a program of type B with a parameter of type A. Such a program may raise an exception, by mapping some $a \in A$ to an exception $e \in E$. In order to catch an exception, it should also be possible to map some $e \in E$ to a non-exceptional value $b \in B$. We formalize this property by choosing the second alternative in Moggi's discussion: programs of type B with a parameter of type A are interpreted by morphisms with codomain TB *and with domain* TA, where the elements of TA are seen as computations of type A rather than values of type TA. This example enlightens one of the reasons why we generalize Moggi's approach. What is kept, and even emphasized, is the distinction between several kinds of programs. In fact, for states as well as for exceptions, we distinguish three kinds of programs, and moreover two kinds of equations. A computational effect is seen as an *apparent lack of soundness*: the intended denotational semantics is not sound, in the sense that it does not satisfy the given axioms, however it becomes sound when some additional information is given.

In order to focus on the effects, our study of states and exceptions is based on a very simple logic: the monadic equational logic. First we provide a detailed description of the intended denotational *semantics* of states and exceptions, using explicitly a set of states and a set of exceptions (claims 1.1 and 1.5). The duality between states and exceptions derives in an obvious way from our presentation (proposition 1.6). It is a duality between the lookup and update operations for states, on one hand, and the *key* throwing and catching operations for exceptions, on the other hand. The key part in throwing an exception is the mapping of some non-exceptional value to an exception, while the key part in catching an exception is the mapping of some exception to a non-exceptional value. Then these key operations have to be encapsulated in order to get the usual raising and handling of exceptions: handling exceptions is obtained by encapsulating the key catching operation inside conditionals. Then we describe the *syntax* of states and exceptions. The computational effects lie in the fact that this syntax does not mention any "type of states" or "type of exceptions", respectively. There are two variants for this syntax: the intended semantics is not a model of the *apparent syntax*, but this lack of soundness is fixed in the *decorated syntax* by providing some additional information (propositions 3.5 and 4.7). The duality between states and the key part of exceptions holds at the syntax level as a duality of effects (theorem 5.1), from which the duality at the semantics level derives easily. We use three different logics for formalizing each computational effect: the intended semantics is described in the *explicit* logic, the apparent syntax in the *apparent* logic and the decorated syntax in the *decorated* logic. The explicit and apparent logics are "usual" logics; in order to focus on the effects we choose two variants of the monadic equational logic. The framework of *diagrammatic logics* provides a simple description of the three logics, including the "unusual" decorated logic; most importantly, it provides a relevant notion of morphisms for relating these three logics.

The paper is organized as follows. The intended semantics of states and exceptions is given in section 1, and the duality is described at the semantics level. Then a simplified version of the framework of diagrammatic logics for effects is presented in section 2, together with a motivating example in order to introduce the notion of "decoration". Section 3 is devoted to states and section 4 to exceptions. In section 5, the duality is extended to the syntax level. In appendix A, some fundamental properties of states and excep-

tions are proved in the decorated logic. In this paper, the word "apparent" is used in the sense of "seeming" ("appearing as such but not necessarily so").

1 States and exceptions: duality of denotational semantics

In this section, the symmetry between states and exceptions is presented as a duality between their intended denotational semantics (proposition 1.6). The aim of the next sections is to extend this result so as to get a symmetry between the syntax of states and exceptions, considered as computational effects, from which the duality between their semantics can be derived (theorem 5.1). In this section we are dealing with sets and functions; the symbols $\times$ and $\prod$ are used for cartesian products, $+$ and $\sum$ for disjoint unions; cartesian products are *products* in the category of sets and disjoint unions are *sums* or *coproducts* in this category.

1.1 States

Let St denote the set of *states*. Let Loc denote the set of *locations* (also called *variables* or *identifiers*). For each location i, let Val_i denote the set of possible *values* for i. For each $i \in Loc$ there is a *lookup* function $l_i : St \to Val_i$ for reading the value of location i in the given state. In addition, for each $i \in Loc$ there is an *update* function $u_i : Val_i \times St \to St$ for setting the value of location i to the given value, without modifying the values of the other locations in the given state. This is summarized as follows. For each $i \in Loc$ there are:

- a set Val_i (values)
- two functions $l_i : St \to Val_i$ (lookup)
  and $u_i : Val_i \times St \to St$ (update)
- and two equalities

$$\begin{cases} \forall\, a \in Val_i\,,\ \forall\, s \in St\,,\ l_i(u_i(a,s)) = a \\ \forall\, a \in Val_i\,,\ \forall\, s \in St\,,\ l_j(u_i(a,s)) = l_j(s) \quad \text{for every } j \neq i \in Loc \end{cases} \tag{1}$$

Let us assume that $St = \prod_{i \in Loc} Val_i$ with the l_i's as projections. Then two states s and s' are equal if and only if $l_i(s) = l_i(s')$ for each i, and the equalities 1 form a coinductive definition of the functions u_i's.

Claim 1.1. This description provides the intended semantics of states.

In [Plotkin & Power 2002] an equational presentation of states is given, with seven families of equations. In [Melliès 2010] these equations are expressed as follows.

1. *Annihilation lookup-update: reading the value of a location i and then updating the location i with the obtained value is just like doing nothing.*
2. *Interaction lookup-lookup: reading twice the same location loc is the same as reading it once.*
3. *Interaction update-update: storing a value a and then a value a' at the same location i is just like storing the value a' in the location.*
4. *Interaction update-lookup: when one stores a value a in a location i and then reads the location i, one gets the value a.*
5. *Commutation lookup-lookup: The order of reading two different locations i and j does not matter.*
6. *Commutation update-update: the order of storing in two different locations i and j does not matter.*
7. *Commutation update-lookup: the order of storing in a location i and reading in another location j does not matter.*

These equations can be translated in our framework as follows, with $l_{i(2)} : St \to Val_i \times St$ defined by $l_{i(2)}(s) = (l_i(s), s)$ and $prr_{Val_i} : Val_i \times St \to St$ by $prr_{Val_i}(a, s) = s$.

$$
\begin{array}{ll}
(1) & \forall i \in Loc,\ \forall\, s \in St,\ \ u_i(l_{i(2)}(s)) = s \in St \\
(2) & \forall i \in Loc,\ \forall\, s \in St,\ \ l_i(prr_{Val_i}(l_{i(2)}(s))) = l_i(s) \in Val_i \\
(3) & \forall i \in Loc,\ \forall\, s \in St,\ a, a' \in Val_i,\ \ u_i(a', u_i(a, s)) = u_i(a', s) \in St \\
(4) & \forall i \in Loc,\ \forall\, s \in St,\ a \in Val_i,\ \ l_i(u_i(a, s)) = a \in Val_i \\
(5) & \forall i \neq j \in Loc,\ \forall\, s \in St,\ \ (l_i(s), l_j(l_{i(2)}(s))) = (l_i(l_{j\,(2)}(s)), l_j(s)) \in Val_i \times Val_j \\
(6) & \forall i \neq j \in Loc,\ \forall\, s \in St,\ a \in Val_i,\ b \in Val_j,\ \ u_j(b, u_i(a, s)) = u_i(a, u_j(b, s)) \in St \\
(7) & \forall i \neq j \in Loc,\ \forall\, s \in St,\ a \in Val_i,\ \ l_{j\,(2)}(u_i(a, s)) = (l_j(s), u_i(a, s)) \in Val_j \times St
\end{array}
\tag{2}
$$

Proposition 1.2. *Let us assume that $St = \prod_{i \in Loc} Val_i$ with the l_i's as projections. Then equations 1 and 2 are equivalent.*

In fact, we prove that, without the assumption about St, equations 1 are equivalent to equations 2 considered as *observational* equations: two states s and s' are observationaly equivalent when $l_k(s) = l_k(s')$ for each location k. These properties are revisited in proposition 3.6 and in appendix A.

Proof. Equations (2) and (5) follow immediately from $prr_{Val_i}(l_{i(2)}(s)) = s$. Equation (4) is the first equation in 1. Equation (7) is $(l_j(u_i(a, s)), u_i(a, s)) = (l_j(s), u_i(a, s))$, which is equivalent to $l_j(u_i(a, s)) = l_j(s)$: this is the second equation in 1. For the remaining equations (1), (3) and (6), which return states, it is easy to check that by applying l_k to both members and using equations 1 we get the same value in Val_k for each location k. □

1.2 Exceptions

The syntax for exceptions heavily depends on the language. For instance:

- In ML-like languages there are several exception names, called *constructors*; the keywords for raising and handling exceptions are `raise` and `handle`, which are used in syntactic constructions like:
  `raise i a` and `... handle i a => g(a) | j b => h(b) | ...`
  where i, j are exception constructors, a, b are parameters and g, h are functions.
- In `Java` there are several exception *types*; the keywords for raising and handling exceptions are `throw` and `try-catch` which are used in syntactic constructions like:
  `throw new i(a)` and `try { ... } catch (i a) g catch (j b) h ...`
  where i, j are exception types, a, b are parameters and g, h are programs.

In spite of the differences in the syntax, the semantics of exceptions is rather similar in many languages. A major point is that there are two kinds of values: the ordinary (i.e., 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.

First let us focus on the raising of exceptions. Let *Exc* denote the set of *exceptions*. Let *ExCstr* denote the set of *exception constructors*. For each exception constructor i, there is a set of *parameters* Par_i and a function $t_i : Par_i \to Exc$ for building the exception $t_i(a)$ of constructor i with the given parameter $a \in Par_i$, called the *key throwing* function. Then the function $raise_{i,Y} : Par_i \to Y + Exc$ for *raising* (or *throwing*) an exception of constructor i into a type Y is made of the key throwing function t_i followed by the inclusion $inr_Y : Exc \to Y + Exc$.

$$raise_{i,Y} \;=\; throw_{i,Y} \;=\; inr_Y \circ t_i \;:\; Par_i \to Y + Exc$$

$$
\begin{array}{ccc}
Par_i & \xrightarrow{\quad raise_{i,Y} \quad} & Y + Exc \\
& \searrow^{t_i} \quad = & \uparrow{\scriptstyle inr} \\
& & Exc
\end{array}
\tag{3}
$$

Claim 1.3. The function $t_i : Par_i \to Exc$ is the *key* function for *throwing* an exception: in the construction of the raising function ($raise_{i,Y}$), only t_i turns a non-exceptional value $a \in Par_i$ to an exception $t_i(a) \in Exc$.

Given a function $f : X \to Y + Exc$ and an element $x \in X$, if $f(x) = raise_{i,Y}(a) \in Y + Exc$ for some $a \in Par_i$ then one says that $f(x)$ *raises an exception of constructor i with parameter a into Y*. One says that a function $f : X + Exc \to Y + Exc$ *propagates exceptions* when it is the identity on Exc. Clearly, any function $f : X \to Y + Exc$ can be *extended by propagating exceptions*: the extended function $Ppg(f) : X + Exc \to Y + Exc$ coincides with f on X and with the identity on Exc.

Now let us study the handling of exceptions, starting from its description in `Java` [Java, 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 $i \in ExCstr$ (point 2). Thus, for handling exceptions of constructors $i_1, \ldots, i_n$ raised by some function $f : X \to Y + Exc$, using functions $g_1 : Par_{i_1} \to Y + Exc, \ldots, g_n : Par_{i_n} \to Y + Exc$, for every $n \geq 1$, the handling process builds a function:

$$f \ handle \ i_1 \Rightarrow g_1 \mid \ldots \mid i_n \Rightarrow g_n \ = \ try\{f\} \ catch \, i_1 \, \{g_1\} \ catch \, i_2 \, \{g_2\} \ldots catch \, i_n \, \{g_n\}$$

which may be seen, equivalently, either as a function from X to $Y + Exc$ or as a function from $X + Exc$ to $Y + Exc$ which propagates the exceptions. We choose the second case, and we use compact notations:

$$f \ handle \ (i_k \Rightarrow g_k)_{1 \leq k \leq n} \ = \ try\{f\} \ catch \, i_k \{g_k\}_{1 \leq k \leq n} : X + Exc \to Y + Exc$$

This function can be defined as follows.

For each $x \in X + Exc$, $(f \ handle \ (i_k \Rightarrow g_k)_{1 \leq k \leq n})(x) \in Y + Exc$ is defined by:
  if $x \in Exc$ then return $x \in Exc \subseteq Y + Exc$;
  // now x is not an exception
  compute $y := f(x) \in Y + Exc$;
  if $y \in Y$ then return $y \in Y \subseteq Y + Exc$;
  // now y is an exception
  for $k = 1..n$ repeat
    if $y = t_{i_k}(a)$ for some $a \in Par_{i_k}$ then return $g_k(a) \in Y + Exc$;
  // now y is an exception not constructed from any $i \in \{i_1, \ldots, i_n\}$
  return $y \in Exc \subseteq Y + Exc$.

In order to express more clearly the apparition of the parameter a when y is an exception of constructor i_k, we introduce for each $i \in ExCstr$ the function $c_i : Exc \to Par_i + Exc$, called the *key catching* function, defined as follows:

For each $e \in Exc$, $c_i(e) \in Par_i + Exc$ is defined by:
if $e = t_i(a)$ then return $a \in Par_i \subseteq Par_i + Exc$;
// *now e is an exception not constructed from i*
return $e \in Exc \subseteq Par_i + Exc$.

This means that the function c_i tests whether the given exception e has constructor i, if so then it *catches the exception* by returning the parameter $a \in Par_i$ such that $e = t_i(a)$, otherwise c_i propagates the exception e. Using the key catching function c_i, the definition of the handling function can be re-stated as follows, with the three embedded conditionals numerated from the innermost to the outermost, for future use.

For each $x \in X + Exc$, $(f \textit{ handle } (i_k \Rightarrow g_k)_{1 \le k \le n})(x) \in Y + Exc$ is defined by:
(3) if $x \in Exc$ then return $x \in Exc \subseteq Y + Exc$;
// *now x is not an exception*
compute $y := f(x) \in Y + Exc$;
(2) if $y \in Y$ then return $y \in Y \subseteq Y + Exc$;
// *now y is an exception*
for $k = 1..n$ repeat
compute $y := c_{i_k}(y) \in Par_{i_k} + Exc$;
(1) if $y \in Par_{i_k}$ then return $g_k(y) \in Y + Exc$;
// *now y is an exception not constructed from any* $i \in \{i_1, \ldots, i_n\}$
return $y \in Exc \subseteq Y + Exc$.

Note that whenever several i's are equal in $(i_1, \ldots, i_n)$, then only the first g_i may be used.

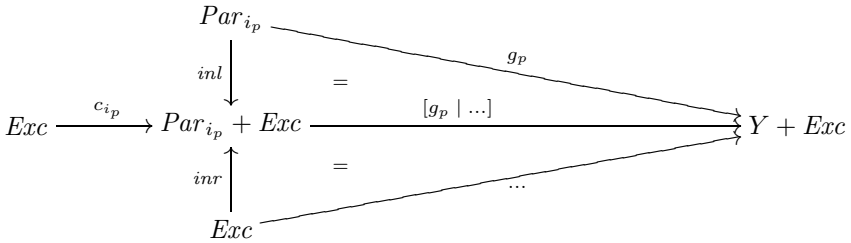
Claim 1.4. The function $c_i : Exc \to Par_i + Exc$ is the *key* function for *catching* an exception: in the construction of the handling function $(f \textit{ handle } i \Rightarrow g)$, only c_i may turn an exception $e \in Exc$ to a non-exceptional value $c_i(e) \in Par_i$, the other parts of the construction propagate all exceptions.

The definition of the handling function is illustrated by the following diagrams; each diagram corresponds to one of the three nested conditionals, from the innermost to the outermost. The inclusions are denoted by $inl_A : A \to A + Exc$ and $inr_A : Exc \to A + Exc$ (subscripts may be dropped) and for every $a : A \to B$ and $e : Exc \to B$ the corresponding conditional is denoted by $[a \,|\, e] : A + Exc \to B$, it is characterized by the equalities $[a \,|\, e] \circ inl_A = a$ and $[a \,|\, e] \circ inr_A = e$.

1. The catching functions $catch\, i_k\{g_k\}_{p \le k \le n} : Exc \to Y + Exc$ are defined recursively by

$$catch\, i_k\{g_k\}_{p \le k \le n} \;=\; \begin{cases} [g_n \mid inr_Y] \circ c_{i_n} & \text{when } p = n \\ [g_p \mid catch\, i_k\{g_k\}_{p+1 \le k \le n}] \circ c_{i_p} & \text{when } p < n \end{cases}$$

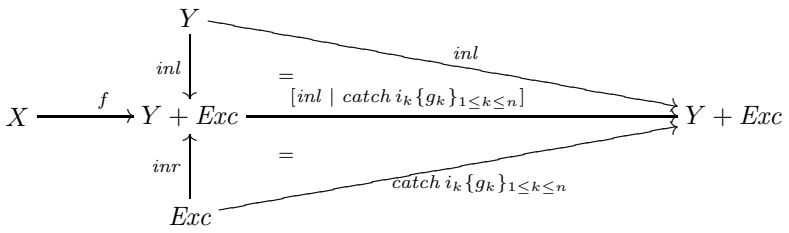
(4)



where ... stands for inr_Y when $p = n$ and for $catch\, i_k\{g_k\}_{p+1 \le k \le n}$ when $p < n$.

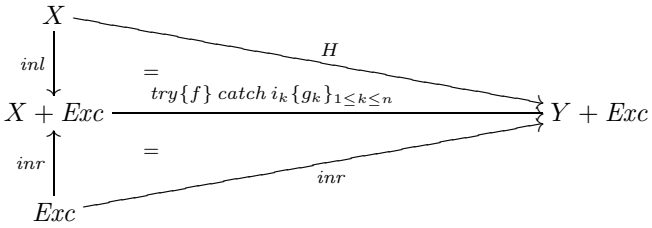
2. Then the function $H : X \to Y + Exc$, which defines the handling function on non-exceptional values, is defined as

$$H \;=\; [inl_Y \mid catch\, i_k\{g_k\}_{1\le k\le n}] \circ f : X \to Y + Exc \tag{5}$$



3. Finally the handling function is the extension of H which propagates exceptions

$$try\{f\}\, catch\, i_k\{g_k\}_{1\le k\le n} \;=\; [H \mid inr_Y] \tag{6}$$



The next claim is based on our previous analysis of `Java` exceptions; it is also related to the notion of *monadic reflection* in [Filinski 1994].

Claim 1.5. This description provides the intended semantics of exceptions.

Let us come back to the key operations t_i and c_i for throwing and catching exceptions. For each $i \in ExCstr$ there are:

- a set Par_i (parameters)
- two functions $t_i : Par_i \to Exc$ (key throwing)
  and $c_i : Exc \to Par_i + Exc$ (key catching)
- and two equalities

$$\begin{cases} \forall\, a \in Par_i\,,\; c_i(t_i(a)) = a \in Par_i \subseteq Par_i + Exc \\ \forall\, b \in Par_j\,,\; c_i(t_j(b)) = t_j(b) \in Exc \subseteq Par_i + Exc \quad \text{for every } j \neq i \in Loc \end{cases} \tag{7}$$

This means that, given an exception e of the form $t_i(a)$, the corresponding key catcher c_i recovers the non-exceptional value a while the other key catchers propagate the exception e. Let us assume that $Exc = \sum_{i\in ExCstr} Par_i$ with the t_i's as coprojections. Then the equalities 7 form an inductive definition of the functions c_i's.

1.3 States and exceptions: the duality

Figure 1 recapitulates the properties of the functions *lookup* (l_i) and *update* (u_i) for states on the left, and the functions *key throw* (t_i) and *key catch* (c_i) for exceptions on the right. Intuitively: for looking up the value of a location i, only the *previous* updating of this location is necessary, and dually, when throwing an exception of constructor i only the *next* catcher for this constructor is necessary (see section 5.2). The next result follows immediately from figure 1.

States	Exceptions
$i \in Loc,\ Val_i,$ $St\,(= \prod_{i \in Loc} Val_i)$ cartesian products: $Val_i \xleftarrow{prl_i} Val_i \times St \xrightarrow{prr_i} St$	$i \in ExCstr,\ Par_i,$ $Exc\,(= \sum_{i \in ExCstr} Par_i)$ disjoint unions: $Exc \xrightarrow{inr_i} Par_i + Exc \xleftarrow{inl_i} Par_i$
$l_i : St \to Val_i$ $u_i : Val_i \times St \to St$	$Exc \leftarrow Par_i : t_i$ $Par_i + Exc \leftarrow Exc : c_i$
$\begin{array}{ccc} Val_i \times St & \xrightarrow{prl_i} & Val_i \\ {\scriptstyle u_i}\downarrow & = & \downarrow{\scriptstyle id} \\ St & \xrightarrow[l_i]{} & Val_i \end{array}$ $\begin{array}{ccccc} Val_i \times St & \xrightarrow{prr_i} & St & \xrightarrow{l_j} & Val_j \\ {\scriptstyle u_i}\downarrow & = & & & \downarrow{\scriptstyle id} \\ St & & \xrightarrow[l_j]{} & & Val_j \end{array}$ $(j \neq i)$	$\begin{array}{ccc} Par_i + Exc & \xleftarrow{inl_i} & Par_i \\ {\scriptstyle c_i}\uparrow & = & \uparrow{\scriptstyle id} \\ Exc & \xleftarrow[t_i]{} & Par_i \end{array}$ $\begin{array}{ccccc} Par_i + Exc & \xleftarrow{inr_i} & Exc & \xleftarrow{t_j} & Par_j \\ {\scriptstyle c_i}\uparrow & = & & & \uparrow{\scriptstyle id} \\ Exc & & \xleftarrow[t_j]{} & & Par_j \end{array}$ $(j \neq i)$

Figure 1: Duality of semantics

Proposition 1.6. *The well-known duality between categorical products and coproducts can be extended as a duality between the semantics of the lookup and update functions for states on one side and the semantics of the key throwing and catching functions for exceptions on the other.*

It would be unfair to consider states and exceptions only from this denotational point of view. Indeed, states and exceptions are *computational effects*, which do not appear explicitly in the syntax: in an imperative language there is no type of states, and in a language with exceptions the type of exceptions that may be raised by a program is not seen as a return type for this program. In fact, our result (theorem 5.1) is that there is a duality between states and exceptions considered as computational effects, which provides the above duality (propostion 1.6) between their semantics.

2 Computational effects

In sections 3 and 4 we will deal with states and exceptions as computational effects. In this section, we present our point of view on computational effects. First a motivating example from object-oriented programming is given, then a simplified version of the framework of diagrammatic logics is presented, and finally this framework is applied to effects.

2.1 An example

In this section we use a toy example dealing with the state of an object in an object-oriented language, in order to outline our approach of computational effects. Let us build a class `BankAccount` for managing (very simple!) bank accounts. We use the types `int` and `void`, and we assume that `int` is interpreted as the set of integers $\mathbb{Z}$ and `void` as a singleton $\{\star\}$. In the class `BankAccount`, there is a method `balance()` which returns the current balance of the account and a method `deposit(x)` for the deposit of `x` Euros on the account. The `deposit` method is a *modifier*, which means that it can use and modify the state of the current account. The `balance` method is an *inspector*, or an *accessor*, which means that it can use the state of the

current account but it is not allowed to modify this state. In the object-oriented language C++, a method is called a *member function*; by default a member function is a modifier, when it is an accessor it is called a *constant member function* and the keyword `const` is used. So, the C++ syntax for declaring the member functions of the class `BankAccount` looks like:

```
int balance ( ) const ;
void deposit (int) ;
```

Forgetting the keyword `const`, this piece of C++ syntax can be translated as a signature $\Sigma_{\text{bank,app}}$, which we call the *apparent signature*:

$$\Sigma_{\text{bank,app}} : \begin{cases} \texttt{balance} : \texttt{void} \to \texttt{int} \\ \texttt{deposit} : \texttt{int} \to \texttt{void} \end{cases} \tag{8}$$

In a model (or algebra) of the signature $\Sigma_{\text{bank,app}}$, the operations would be interpreted as functions:

$$\begin{cases} [[\texttt{balance}]] : \{\star\} \to \mathbb{Z} \\ [[\texttt{deposit}]] : \mathbb{Z} \to \{\star\} \end{cases}$$

which clearly is not the intended interpretation.

In order to get the right semantics, we may use another signature $\Sigma_{\text{bank,expl}}$, which we call the *explicit signature*, with a new symbol `state` for the "type of states":

$$\Sigma_{\text{bank,expl}} : \begin{cases} \texttt{balance} : \texttt{state} \to \texttt{int} \\ \texttt{deposit} : \texttt{int} \times \texttt{state} \to \texttt{state} \end{cases} \tag{9}$$

The intended interpretation is a model of the explicit signature $\Sigma_{\text{bank,expl}}$, with St denoting the set of states of a bank account:

$$\begin{cases} [[\texttt{balance}]] : St \to \mathbb{Z} \\ [[\texttt{deposit}]] : \mathbb{Z} \times St \to St \end{cases}$$

So far, in this example, we have considered two different signatures. On the one hand, the apparent signature $\Sigma_{\text{bank,app}}$ is simple and quite close to the C++ code, but the intended semantics is not a model of $\Sigma_{\text{bank,app}}$. On the other hand, the semantics is a model of the explicit signature $\Sigma_{\text{bank,expl}}$, but $\Sigma_{\text{bank,expl}}$ is far from the C++ syntax: actually, the very nature of the object-oriented language is lost by introducing a "type of states". Let us now define a *decorated signature* $\Sigma_{\text{bank,deco}}$, which is still closer to the C++ code than the apparent signature and which has a model corresponding to the intended semantics. The decorated signature is not exactly a signature in the classical sense, because there is a classification of its operations. This classification is provided by superscripts called *decorations*: the decorations "`(1)`" and "`(2)`" correspond respectively to the object-oriented notions of *accessor* and *modifier*.

$$\Sigma_{\text{bank,deco}} : \begin{cases} \texttt{balance}^{(1)} : \texttt{void} \to \texttt{int} \\ \texttt{deposit}^{(2)} : \texttt{int} \to \texttt{void} \end{cases} \tag{10}$$

The decorated signature is similar to the C++ code, with the decoration "`(1)`" corresponding to the keyword "`const`". In addition, we claim that the intended semantics can be seen as a *decorated model* of this decorated signature.

In order to add to the signature the constants of type `int` like `0`, `1`, `2`, ... and the usual operations on integers, a third decoration is used: the decoration "`(0)`" for *pure* functions, which means, for functions which neither inspect nor modify the state of the bank account. So, we add to the apparent and explicit signatures the constants `0`, `1`, ...: `void` $\to$ `int` and the operations `+`, `-`, $*$: `int` $\times$ `int` $\to$ `int`, and we add to the decorated signature the pure constants $\texttt{0}^{(0)}$, $\texttt{1}^{(0)}$, ...: `void` $\to$ `int` and the pure operations $\texttt{+}^{(0)}$, $\texttt{-}^{(0)}$, $*^{(0)}$: `int` $\times$ `int` $\to$ `int`. For instance in the C++ expressions

```
deposit(7); balance()  and  7 + balance()
```

composition is expressed in several different ways: in the functional way $f(a)$, in the infix way $a\ f\ b$ and in the imperative way $c; c'$. In the explicit signature, these expressions can be seen as the terms $\texttt{balance} \circ \texttt{deposit} \circ (7 \times \texttt{id}_{\texttt{state}})$ and $\texttt{+} \circ (7 \times \texttt{balance})$, with $\texttt{void} \times \texttt{state}$ identified with $\texttt{state}$:

$$\texttt{state} \simeq \texttt{void} \times \texttt{state} \xrightarrow{7 \times \texttt{id}_{\texttt{state}}} \texttt{int} \times \texttt{state} \xrightarrow{\texttt{deposit}} \texttt{state} \xrightarrow{\texttt{balance}} \texttt{int}$$

$$\texttt{state} \simeq \texttt{void} \times \texttt{state} \xrightarrow{7 \times \texttt{balance}} \texttt{int} \times \texttt{int} \xrightarrow{\texttt{+}} \texttt{int}$$

In the decorated signature, they can be seen as the decorated terms $\texttt{balance}^{(1)} \circ \texttt{deposit}^{(2)} \circ 7^{(0)}$ and $\texttt{+}^{(0)} \circ \langle 7^{(0)}, \texttt{balance}^{(1)} \rangle$:

$$\texttt{void} \xrightarrow{7^{(0)}} \texttt{int} \xrightarrow{\texttt{deposit}^{(2)}} \texttt{void} \xrightarrow{\texttt{balance}^{(1)}} \texttt{int}$$

$$\texttt{void} \xrightarrow{\langle 7^{(0)}, \texttt{balance}^{(1)} \rangle} \texttt{int} \times \texttt{int} \xrightarrow{\texttt{+}^{(0)}} \texttt{int}$$

These two expressions have different effects: the first one is a modifier while the second one is an accessor; however, both return the same result (an integer). We introduce the symbol $\sim$ for the relation "same result, maybe distinct effects"; the relation $\sim$ will be considered as a decorated version of the equality.

$$\texttt{balance}^{(1)} \circ \texttt{deposit}^{(2)} \circ 7^{(0)} \ \sim\ \texttt{+}^{(0)} \circ \langle 7^{(0)}, \texttt{balance}^{(1)} \rangle$$

2.2 Simplified diagrammatic logics

In this paper, as in [Domínguez & Duval 2010] and [Dumas et al. 2011], we use the point of view of *diagrammatic logics* for dealing with computational effects. One fundamental feature of the theory of diagrammatic logics is the distinction between a logical theory and its presentations (or specifications). This is the usual point of view in the framework of algebraic specifications [Ehrig & Mahr 1985], but not always in logic, as mentioned by F.W. Lawvere in his foreword to [Adámek et al. 2011]: *Yet many works in general algebra (and model theory generally) continue anachronistically to confuse a presentation in terms of signatures with the presented theory itself.* A second fundamental feature of the theory of diagrammatic logics is the definition of a rich family of morphisms of logics. Computational effects, from our point of view, heavily depend on some morphisms of logics. Thus, in this paper, in order to focus on states and exceptions as effects, we use a simplified version of diagrammatic logics by dropping the distinction between a logical theory and its presentations. It is only in remark 2.9 that we give some hints about non-simplified diagrammatic logics.

On the other hand, with the same goal of focusing on states and exceptions as effects, in sections 3 and 4 the base logic is the very simple (multi-sorted) *monadic equational logic*, where a theory is made of types, unary terms and equations. We will occasionally mention the *equational logic*, where in addition a theory may have terms of any finite arity. In order to keep the syntactic aspect of the logics, we use a congruence relation between terms rather than the equality; in the denotational semantics, this congruence is usually interpreted as the equality.

Definition 2.1. A *simplified diagrammatic logic* is a category $\mathbf{T}$ with colimits; its objects are called the $\mathbf{T}$-*theories* and its morphisms the *morphisms of* $\mathbf{T}$-*theories*. A *morphism of simplified diagrammatic logics* $F : \mathbf{T} \to \mathbf{T}'$ is a left adjoint functor. This yields the *category of simplified diagrammatic logics*.

Example 2.2 (Monadic equational logic)**.** A monadic equational theory might be called a "syntactic category": it 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 *objects* or *types* and its edges are called *morphisms* or *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, denoted by $\equiv$, which is a *congruence* with respect to the composition and such that the associativity and identity axioms hold up to congruence. This definition of the monadic equational logic can be described by a set of *inference rules*, as in figure 2. A morphism of monadic equational theories might be called a "syntactic functor": it maps types to types, terms to terms and equations to equations.

$$\text{(comp)}\ \frac{f : X \to Y \quad g : Y \to Z}{g \circ f : X \to Z} \qquad \text{(id)}\ \frac{X}{id_X : X \to X}$$

$$\text{(assoc)}\ \frac{f : X \to Y \quad g : Y \to Z \quad h : Z \to W}{h \circ (g \circ f) \equiv (h \circ g) \circ f}$$

$$\text{(id-src)}\ \frac{f : X \to Y}{f \circ id_X \equiv f} \qquad \text{(id-tgt)}\ \frac{f : X \to Y}{id_Y \circ f \equiv f}$$

$$\text{(}\equiv\text{-refl)}\ \frac{}{f \equiv f} \qquad \text{(}\equiv\text{-sym)}\ \frac{f \equiv g}{g \equiv f} \qquad \text{(}\equiv\text{-trans)}\ \frac{f \equiv g \quad g \equiv h}{f \equiv h}$$

$$\text{(}\equiv\text{-subs)}\ \frac{f : X \to Y \quad g_1 \equiv g_2 : Y \to Z}{g_1 \circ f \equiv g_2 \circ f : X \to Z}$$

$$\text{(}\equiv\text{-repl)}\ \frac{f_1 \equiv f_2 : X \to Y \quad g : Y \to Z}{g \circ f_1 \equiv g \circ f_2 : X \to Z}$$

Figure 2: Rules of the monadic equational logic

Example 2.3 (Equational logic)**.** An equational theory might be called a "syntactic category with finite products". Precisely, an *equational theory* is a monadic equational theory with in addition, for each finite family $(Y_i)_{1 \leq i \leq n}$ of types, a *product (up to congruence)* made of a cone $(q_i : \prod_{j=1}^n Y_j \to Y_i)_{1 \leq i \leq n}$ such that for each cone $(f_i : X \to Y_i)_{1 \leq i \leq n}$ with the same base there is a term $\langle f_1, \ldots, f_n \rangle : X \to \prod_{j=1}^n Y_j$ such that $q_i \circ \langle f_1, \ldots, f_n \rangle \equiv f_i$ for each i, and whenever some $g : X \to \prod_{j=1}^n Y_j$ is such that $q_i \circ g \equiv f_i$ for each i then $g \equiv \langle f_1, \ldots, f_n \rangle$. When $n = 0$ this means that in an equational theory there is a *terminal* type $\mathbb{1}$ such that for each type X there is a term $\langle\,\rangle_X : X \to \mathbb{1}$, which is unique up to congruence in the sense that every $g : X \to \mathbb{1}$ satisfies $g \equiv \langle\,\rangle_X$. A morphism of equational theories is a morphism of monadic equational theories which preserves products. This definition can be described by a set of *inference rules*, as in figure 3. When there are several parts in the conclusion of a rule, this must be understood as a conjunction (which might be avoided by writing several rules). The monadic equational logic may be seen as the restriction of the equational logic to terms with exactly one "variable". The functor which maps each monadic equational theory to its generated equational theory is a morphism of simplified diagrammatic logics, with right adjoint the forgetful functor.

Given a simplified diagrammatic logic, we define the associated notions of model and inference system. We often write "logic" instead of "simplified diagrammatic logic".

Definition 2.4. Let $\mathbf{T}$ be a logic. Let Φ and Θ be $\mathbf{T}$-theories, a *model* of Φ in Θ is a morphism from Φ to Θ in $\mathbf{T}$. Then the triple $\Lambda = (\Phi, \Theta, M)$ is a *language* on $\mathbf{T}$ with *syntax* Φ and *semantics* M. The set of models of Φ in Θ is denoted by $\mathrm{Mod}_{\mathbf{T}}(\Phi, \Theta)$.

Remark 2.5. The definitions are such that every simplified diagrammatic logic $\mathbf{T}$ has the *soundness* property: in every language, the semantics is a model of the syntax.

Definition 2.6. Let $\mathbf{T}$ be a logic. An *inference rule* is a morphism $\rho : \mathcal{C} \to \mathcal{H}$ in $\mathbf{T}$. Then $\mathcal{H}$ is the *hypothesis* and $\mathcal{C}$ is the *conclusion* of the rule ρ. Let Φ_0 and Φ be $\mathbf{T}$-theories, an *instance* of Φ_0 in Φ is a morphism $\kappa : \Phi_0 \to \Phi$ in $\mathbf{T}$. The *inference step* applying a rule $\rho : \mathcal{C} \to \mathcal{H}$ to an instance $\kappa : \mathcal{H} \to \Phi$ of $\mathcal{H}$ in Φ is the composition in $\mathbf{T}$, which builds the instance $\kappa \circ \rho : \mathcal{C} \to \Phi$ of $\mathcal{C}$ in Φ.

Rules of the monadic equational logic, and for each $n \in \mathbb{N}$:	i.e., when $n = 0$:
$\dfrac{Y_1 \ \ldots \ Y_n}{(q_i : \prod_{j=1}^n Y_j \to Y_i)_{1 \le i \le n}}$	$\dfrac{}{\overline{\mathbb{1}}}$
$\dfrac{(q_i : \prod_{j=1}^n Y_j \to Y_i)_{1 \le i \le n} \quad (f_i : X \to Y_i)_{1 \le i \le n}}{\langle f_1, \ldots, f_n \rangle : X \to \prod_{j=1}^n Y_j \quad \forall i \ q_i \circ \langle f_1, \ldots, f_n \rangle \equiv f_i}$	$\dfrac{X}{\langle\,\rangle_X : X \to \mathbb{1}}$
$\dfrac{(q_i : \prod_{j=1}^n Y_j \to Y_i)_{1 \le i \le n} \quad g : X \to \prod_{j=1}^n Y_j \quad \forall i \ q_i \circ g \equiv f_i}{g \equiv \langle f_1, \ldots, f_n \rangle}$	$\dfrac{g : X \to \mathbb{1}}{g \equiv \langle\,\rangle_X}$

Figure 3: Rules of the equational logic

Remark 2.7. The rule $\rho : \mathcal{C} \to \mathcal{H}$ may be represented in the usual way as a "fraction" $\frac{\mathcal{H}}{\rho(\mathcal{C})}$, or as $\frac{\mathcal{H}_1,\ldots,\mathcal{H}_k}{\rho(\mathcal{C})}$ when $\mathcal{H}$ is the colimit of several theories, see example 2.8. In addition, in [Domínguez & Duval 2010] it is explained why an inference rule written in the usual way as a "fraction" $\frac{\mathcal{H}}{\rho(\mathcal{C})}$ is really a *fraction* in the categorical sense of [Gabriel & Zisman 1967], but with $\mathcal{H}$ on the denominator side and $\mathcal{C}$ on the numerator side.

Example 2.8 (Composition rule)**.** Let us consider the equational logic $\mathbf{T}_{\text{eq}}$, as in example 2.3. The category of sets can be seen as an equational theory Θ_{set}, with the equalities as equations and the cartesian products as products. Let us define the equational theory "of integers" Φ_{int} as the equational theory generated by a type I, three terms $z : \mathbb{1} \to I$ and $s, p : I \to I$ and two equations $s \circ p \equiv id_I$ and $p \circ s \equiv id_I$. Then there is a unique model M_{int} of Φ_{int} in Θ_{set} which interprets the sort I as the set $\mathbb{Z}$ of integers, the constant term z as 0 and the terms s and p as the functions $x \mapsto x + 1$ and $x \mapsto x - 1$. In the equational logic $\mathbf{T}_{\text{eq}}$, let us consider the composition rule:

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

Let $\mathcal{H}$ be the equational theory generated by three types X, Y, Z and two consecutive terms $f : X \to Y$, $g : Y \to Z$; let $\mathcal{C}$ be the equational theory generated by two types T, T' and a term $t : T \to T'$. The composition rule corresponds to the morphism of equational theories from $\mathcal{C}$ to $\mathcal{H}$ which maps t to $g \circ f$. Let us consider the instance κ of $\mathcal{H}$ in Φ_{int} which maps f and g respectively to z and s, then the inference step applying the composition rule to this instance κ builds the instance of $\mathcal{C}$ in Φ_{int} which maps t to $s \circ z$, as required. Moreover, $\mathcal{H}$ can be obtained as the pushout of $\mathcal{H}_1$ (generated by X, Y and $f : X \to Y$) and $\mathcal{H}_2$ (generated by Y, Z and $g : Y \to Z$) on their common part (the equational theory generated by Y). Then the instance κ of $\mathcal{H}$ in Φ_{int} can be built from the instance κ_1 of $\mathcal{H}_1$ in Φ_{int} mapping f to z and the instance κ_2 of $\mathcal{H}_2$ in Φ_{int} mapping g to s.

Remark 2.9. In this simplified version of diagrammatic logic, the morphisms of theories serve for many purposes. However in the non-simplified version there is a distinction between theories and their presentations (called *specifications*), which results in more subtle definitions. This is outlined here, more details can be found in [Domínguez & Duval 2010]. This will not be used in the next sections. As usual a *locally presentable category* is a category $\mathbf{C}$ which is equivalent to the category of set-valued realizations (or models) of a limit sketch [Gabriel & Ulmer 1971]. In addition, a functor $F\colon \mathbf{C}_1 \to \mathbf{C}_2$ which is the left adjoint to the precomposition with some morphism of limit sketches [Ehresmann 1968] will be called a *locally presentable functor*.

- A *diagrammatic logic* is defined as a locally presentable functor $L : \mathbf{S} \to \mathbf{T}$ such that its right adjoint R is full and faithful. This means that L is a *localization*, up to an equivalence of categories: it consists of adding inverse morphisms for some morphisms, constraining them to become isomorphisms [Gabriel & Zisman 1967]. The categories $\mathbf{S}$ and $\mathbf{T}$ are called the category of *specifications* and the category of *theories*, respectively, of the diagrammatic logic L. A specification Σ *presents* a theory Θ if Θ is isomorphic to $L(\Sigma)$. The fact that R is full and faithful means that every theory Θ, when seen as a specification $R(\Theta)$, presents itself.

- A *model* M of a specification Σ in a theory Θ is a morphism of theories $M \colon L\Sigma \to \Theta$ or equivalently, thanks to the adjunction, a morphism of specifications $M \colon \Sigma \to R\Theta$.

- An *entailment* is a morphism τ in $\mathbf{S}$ such that $L\tau$ is invertible in $\mathbf{T}$; a similar notion can be found in [Makkai 1997]. An *instance* κ of a specification Σ_0 in a specification Σ is a cospan in $\mathbf{S}$ made of a morphism $\sigma : \Sigma_0 \to \Sigma'$ and an entailment $\tau : \Sigma \to \Sigma'$. It is also called a *fraction* with *numerator* σ and *denominator* τ [Gabriel & Zisman 1967]. The instances can be composed in the usual way as cospans, thanks to pushouts in $\mathbf{S}$. This forms the *bicategory of instances* of the logic, and $\mathbf{T}$ is, up to equivalence, the quotient category of this bicategory. An *inference rule* ρ with *hypothesis* $\mathcal{H}$ and *conclusion* $\mathcal{C}$ is an instance of $\mathcal{C}$ in $\mathcal{H}$. Then an inference step is a composition of fractions.

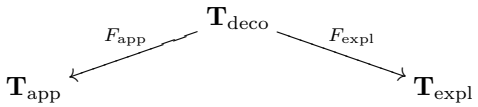
- An *inference system* for a diagrammatic logic L is a morphism of limit sketches which gives rise to the locally presentable functor L. The *elementary* inference rules are the rules in the image of the inference system by the Yoneda contravariant functor. Then a *derivation*, or *proof*, is the description of a fraction in terms of elementary inference rules.

- A *morphism of logics* $F \colon L_1 \to L_2$, where $L_1 : \mathbf{S}_1 \to \mathbf{T}_1$ and $L_2 : \mathbf{S}_2 \to \mathbf{T}_2$, is a pair of locally presentable functors (F_S, F_T) with $F_S : \mathbf{S}_1 \to \mathbf{S}_2$ and $F_T : \mathbf{T}_1 \to \mathbf{T}_2$, together with a natural isomorphism $F_T \circ L_1 \cong L_2 \circ F_S$ induced by a commutative square of limit sketches.

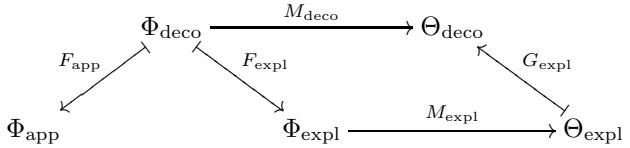
2.3 Diagrammatic logics for effects

Now let us come back to computational effects. Our point of view is that a language with computational effect is a kind of language with an *apparent lack of soundness*: a language with computational effect is made of a syntax, called the *apparent syntax*, and a semantics which (in general) *is not* a model of the apparent syntax, together with some additional information which may be added to the apparent syntax in order to get another syntax, called the *decorated syntax*, such that the semantics *is* a model of the decorated syntax. This approach leads to a new point of view about effects, which can be seen as a generalization of the point of view of *monads*: the distinction between values and computations provided by the monad can be seen as a kind of decoration. In our framework every logic is sound (remark 2.5), and a computational effect is defined with respect to a *span* of logics, which means, a pair of morphisms of logics with the same domain.

Definition 2.10. Let $\mathcal{Z}$ be a span in the category of simplified diagrammatic logics:



We call $\mathbf{T}_{\text{app}}$ the *apparent* logic, $\mathbf{T}_{\text{deco}}$ the *decorated* logic and $\mathbf{T}_{\text{expl}}$ the *explicit* logic. Let G_{expl} denote the right adjoint of F_{expl}. A *language with effect* with respect to $\mathcal{Z}$ is a language $\Lambda_{\text{deco}} = (\Phi_{\text{deco}}, \Theta_{\text{deco}}, M_{\text{deco}})$ in $\mathbf{T}_{\text{deco}}$ together with a theory Θ_{expl} in $\mathbf{T}_{\text{expl}}$ such that $\Theta_{\text{deco}} = G_{\text{expl}}\Theta_{\text{expl}}$. The *apparent syntax* of Λ_{deco} is $\Phi_{\text{app}} = F_{\text{app}}\Phi_{\text{deco}}$ in $\mathbf{T}_{\text{app}}$. The *expansion* of Λ_{deco} is the language $\Lambda_{\text{expl}} = (\Phi_{\text{expl}}, \Theta_{\text{expl}}, M_{\text{expl}})$ in $\mathbf{T}_{\text{expl}}$ with $\Phi_{\text{expl}} = F_{\text{expl}}\Phi_{\text{deco}}$ and $M_{\text{expl}} = \varphi M_{\text{deco}}$, where $\varphi : \text{Mod}_{\mathbf{T}_{\text{deco}}}(\Phi_{\text{deco}}, \Theta_{\text{deco}}) \to \text{Mod}_{\mathbf{T}_{\text{expl}}}(\Phi_{\text{expl}}, \Theta_{\text{expl}})$ is the bijection provided by the adjunction $F_{\text{expl}} \dashv G_{\text{expl}}$.



Remark 2.11. Since a language with effect Λ_{deco} is defined as a language on $\mathbf{T}_{\text{deco}}$, according to remark 2.5 it is *sound*. Similarly, the expansion Λ_{expl} of Λ_{deco} is a language on $\mathbf{T}_{\text{expl}}$, hence it is *sound*. Both languages are equivalent from the point of view of semantics, thanks to the bijection φ. This may be used for formalizing a computational effect when the decorated syntax corresponds to the programs while the explicit syntax does not, as in the bank account example in section 2.1.

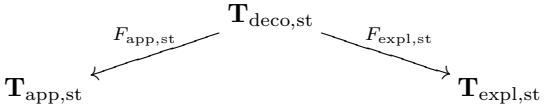
Remark 2.12. It is tempting to look for a language $\Lambda_{\text{app}} = (\Phi_{\text{app}}, \Theta_{\text{app}}, M_{\text{app}})$ on $\mathbf{T}_{\text{app}}$, where $\Phi_{\text{app}} = F_{\text{app}}\Phi_{\text{deco}}$ is the apparent syntax of Λ_{deco}. However, in general such a language does not exist (as for instance in remark 3.4).

3 States

In the syntax of an imperative language there is no type of states (the state is "hidden") while the interpretation of this language involves a set of states St. More precisely, if the types X and Y are interpreted as the sets $[[X]]$ and $[[Y]]$, then each term $f : X \to Y$ is interpreted as a function $[[f]] : [[X]] \times St \to [[Y]] \times St$. In Moggi's papers introducing monads for effects [Moggi 1989, Moggi 1991] such a term $f : X \to Y$ is called a *computation*, and whenever the function $[[f]]$ is $[[f]]_{(0)} \times id_{St}$ for some $[[f]]_{(0)} : [[X]] \to [[Y]]$ then f is called a *value*. We keep this distinction, using *modifier* and *pure term* instead of *computation* and *value*, respectively. In addition, an *accessor* (or *inspector*) is a term $f : X \to Y$ that is interpreted by a function $[[f]] = \langle [[f]]_{(1)}, prr_{[[X]]} \rangle$, for some $[[f]]_{(1)} : [[X]] \times St \to [[Y]]$, where $prr_{[[X]]} : [[X]] \times St \to St$ is the projection. It follows that every pure term is an accessor and every accessor is a modifier. We will use the decorations (0), (1) and (2), written as superscripts, for pure terms, accessors and modifiers, respectively. Moreover, we distinguish two kinds of equations: when $f, g : X \to Y$ are parallel terms, then a *strong* equation $f \equiv g$ is interpreted as the equality $[[f]] = [[g]] : [[X]] \times St \to [[Y]] \times St$, while a *weak* equation $f \sim g$ is interpreted as the equality $prl_{[[Y]]} \circ [[f]] = prl_{[[Y]]} \circ [[g]] : [[X]] \times St \to [[Y]]$, where $prl_{[[Y]]} : [[Y]] \times St \to [[Y]]$ is the projection. Clearly, both notions coincide on accessors, hence on pure terms.

3.1 A span of logics for states

Let *Loc* be a given set, called the set of *locations*. Let us define a span of logics for dealing with states (with respect to the set of locations *Loc*) denoted by $\mathcal{Z}_{\text{st}}$:



In this section the subscript "st" will be omitted. First the decorated logic is defined, then the apparent logic and the morphism F_{app}, and finally the explicit logic and the morphism F_{expl}. For each logic the definition of the morphisms of theories is omitted, since it derives in a natural way from the definition of the theories. In order to focus on the fundamental properties of states as effects, these logics are based on the *monadic equational logic* (as in example 2.2).

The logic $\mathbf{T}_{\text{deco}}$ is the *decorated monadic equational logic for states* (with respect to *Loc*), defined as follows. A theory Θ_{deco} for this logic is made of:

- Three nested monadic equational theories $\Theta^{(0)} \subseteq \Theta^{(1)} \subseteq \Theta^{(2)}$ with the same types, such that the congruence on $\Theta^{(0)}$ and on $\Theta^{(1)}$ is the restriction of the congruence $\equiv$ on $\Theta^{(2)}$. The objects of any of

the three categories are called the *types* of the theory, the terms in $\Theta^{(2)}$ are called the *modifiers*, those in $\Theta^{(1)}$ may be called the *accessors*, and if they are in $\Theta^{(0)}$ they may be called the *pure terms*. The relations $f \equiv g$ are called the *strong equations*.

- An equivalence relation $\sim$ between parallel terms, which satisfies the properties of substitution and pure replacement (defined in figure 4). The relations $f \sim g$ are called the *weak equations*. Every strong equation is a weak equation and every weak equation between accessors is a strong equation.
- A distinguished type $\mathbb{1}$ which has the following *decorated terminality* property: for each type X there is a pure term $\langle\rangle_X : X \to \mathbb{1}$ such that every modifier $f : X \to \mathbb{1}$ satisfies $f \sim \langle\rangle_X$.
- And Θ may have *decorated products on Loc*, where a decorated product on *Loc* is defined as a cone of accessors $(q_i : Y \to Y_i)_{i \in Loc}$ such that for each cone of accessors $(f_i : X \to Y_i)_{i \in Loc}$ with the same base there is a modifier $\langle f_j \rangle_{j \in Loc} : X \to Y$ such that $q_i \circ \langle f_j \rangle_{j \in Loc} \sim f_i$ for each i, and whenever some modifier $g : X \to Y$ is such that $q_i \circ g \sim f_i$ for each i then $g \equiv \langle f_j \rangle_{j \in Loc}$.

Figure 4 provides the *decorated rules* for states, which describe the properties of the decorated theories. We use the following conventions: $X, Y, Z, \ldots$ are types, $f, g, h, \ldots$ are terms, $f^{(0)}$ means that f is a pure term, $f^{(1)}$ means that f is an accessor, and similarly $f^{(2)}$ means that f is a modifier (this is always the case but the decoration may be used for emphasizing). Decoration hypotheses may be grouped with other hypotheses: for instance, "$f^{(1)} \sim g^{(1)}$" means "$f^{(1)}$ and $g^{(1)}$ and $f \sim g$". A decorated product on *Loc* is denoted by $(q_i^{(1)} : \prod_j Y_j \to Y_i)_i$.

Remark 3.1. There is no general replacement rule for weak equations: if $f_1 \sim f_2 : X \to Y$ and $g : Y \to Z$ then in general $g \circ f_1 \not\sim g \circ f_2$, except when g is pure.

Example 3.2. Let us derive the following rule, which says that $\langle\rangle_X$ is the unique accessor from X to $\mathbb{1}$, up to strong equations:

$$(\equiv\text{-final})\ \frac{f^{(1)} : X \to \mathbb{1}}{f \equiv \langle\rangle_X}$$

The derivation tree is:

$$(\sim\text{-to-}\equiv)\ \frac{f^{(1)} \qquad (\sim\text{-final})\ \dfrac{f : X \to \mathbb{1}}{f \sim \langle\rangle_X} \qquad (\text{0-to-1})\ \dfrac{(\text{0-final})\ \dfrac{X}{\langle\rangle_X^{(0)}}}{\langle\rangle_X^{(1)}}}{f \equiv \langle\rangle_X}$$

Now let us describe the "apparent" side of the span. The logic $\mathbf{T}_{\text{app}}$ extends the monadic equational logic as follows : a theory of $\mathbf{T}_{\text{app}}$ is a monadic equational theory with a terminal object $\mathbb{1}$ which may have products on *Loc* (i.e., with their base indexed by *Loc*). The morphism $F_{\text{app}} : \mathbf{T}_{\text{deco}} \to \mathbf{T}_{\text{app}}$ maps each theory Θ_{deco} of $\mathbf{T}_{\text{deco}}$ to the theory Θ_{app} of $\mathbf{T}_{\text{app}}$ made of:

- A type $\widehat{X}$ for each type X in Θ_{deco}.
- A term $\widehat{f} : \widehat{X} \to \widehat{Y}$ for each modifier $f : X \to Y$ in Θ_{deco} (which includes the accessors and the pure terms), such that $\widehat{id_X} = id_{\widehat{X}}$ for each type X and $\widehat{g \circ f} = \widehat{g} \circ \widehat{f}$ for each pair of consecutive modifiers (f, g).
- An equation $\widehat{f} \equiv \widehat{g}$ for each weak equation $f \sim g$ in Θ_{deco} (which includes the strong equations).
- A product $(\widehat{q_i} : \prod_j \widehat{Y_j} \to \widehat{Y_i})_{i \in Loc}$ for each decorated product $(q_i^{(1)} : \prod_j Y_j \to Y_i)_{i \in Loc}$ in Θ_{deco}.

Rules of the monadic equational logic, and:

$$\text{(0-to-1)}\ \frac{f^{(0)}}{f^{(1)}} \qquad \text{(1-to-2)}\ \frac{f^{(1)}}{f^{(2)}}$$

$$\text{(0-comp)}\ \frac{f^{(1)} \quad g^{(1)}}{(g \circ f)^{(1)}} \qquad \text{(1-comp)}\ \frac{f^{(0)} \quad g^{(0)}}{(g \circ f)^{(0)}} \qquad \text{(0-id)}\ \frac{X}{id_X^{(0)} : X \to X}$$

$$\text{(}\sim\text{-refl)}\ \frac{}{f \sim f} \qquad \text{(}\sim\text{-sym)}\ \frac{f \sim g}{g \sim f} \qquad \text{(}\sim\text{-trans)}\ \frac{f \sim g \quad g \sim h}{f \sim h}$$

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

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

$$\text{(final)}\ \frac{}{\mathbb{1}} \qquad \text{(0-final)}\ \frac{X}{\langle\,\rangle_X^{(0)} : X \to \mathbb{1}} \qquad \text{(}\sim\text{-final)}\ \frac{f : X \to \mathbb{1}}{f \sim \langle\,\rangle_X}$$

$$\text{(tuple)}\ \frac{(q_i^{(1)} : \prod_j Y_j \to Y_i)_i \quad (f_i^{(1)} : X \to Y_i)_i}{\langle f_j \rangle_j : X \to Y \quad \forall i\ q_i \circ \langle f_j \rangle_j \sim f_i}$$

$$\text{(}\equiv\text{-tuple)}\ \frac{(q_i^{(1)} : \prod_j Y_j \to Y_i)_i \quad g : X \to Y \quad \forall i\ q_i \circ g \sim f_i^{(1)}}{g \equiv \langle f_j \rangle_j}$$

Figure 4: Rules of the decorated logic for states

Thus, the morphism F_{app} blurs the distinction between modifiers, accessors and pure terms, as well as the distinction between weak and strong equations. In the following, the notation $\widehat{\ldots}$ will be omitted.

It follows from the definition of F_{app} that each rule of the decorated logic $\mathbf{T}_{\text{deco}}$ is mapped by F_{app} to a rule of the apparent logic $\mathbf{T}_{\text{app}}$, so that F_{app} is a morphism of diagrammatic logics. The morphism F_{app} can be used for checking a decorated proof in two steps, by checking first that its image by F_{app} is a proof in $\mathbf{T}_{\text{app}}$.

Now let us describe the "explicit" side of the span. The logic $\mathbf{T}_{\text{expl}}$ extends the monadic equational logic as follows : a theory of $\mathbf{T}_{\text{expl}}$ is a monadic equational theory with a distinguished object S, called the *type of states*, with a product-with-S functor $X \times S$, and which may have products on *Loc*. The morphism $F_{\text{expl}} : \mathbf{T}_{\text{deco}} \to \mathbf{T}_{\text{expl}}$ maps each theory $\Theta_{\text{deco}} = (\Theta^{(0)} \subseteq \Theta^{(1)} \subseteq \Theta^{(2)})$ of $\mathbf{T}_{\text{deco}}$ to the theory Θ_{expl} of $\mathbf{T}_{\text{expl}}$ made of:

- A type $\widetilde{X}$ for each type X in Θ_{deco}; the projections from $\widetilde{X} \times S$ are denoted by $prl_X : \widetilde{X} \times S \to \widetilde{X}$ and $prr_X : \widetilde{X} \times S \to S$.
- A term $\widetilde{f} : \widetilde{X} \times S \to \widetilde{Y} \times S$ for each modifier $f : X \to Y$ in Θ_{deco}, such that:
  - if in addition f is an accessor then there is a term $\widetilde{f}_{(1)} : \widetilde{X} \times S \to \widetilde{Y}$ such that $\widetilde{f} = \langle \widetilde{f}_{(1)}, prr_X \rangle$,

– and if moreover f is a pure term then there is a term $\widetilde{f}_{(0)} : \widetilde{X} \to \widetilde{Y}$ such that $\widetilde{f}_{(1)} = \widetilde{f}_{(0)} \circ prl_X : \widetilde{X} \times S \to \widetilde{Y}$, hence $\widetilde{f} = \langle \widetilde{f}_{(0)} \circ prl_X, prr_X \rangle = \widetilde{f}_{(0)} \times id_S$.

such that $\widetilde{id_X} = id_{\widetilde{X} \times S}$ for each type X and $\widetilde{g \circ f} = \widetilde{g} \circ \widetilde{f}$ for each pair of consecutive modifiers (f, g).

- An equation $\widetilde{f} \equiv \widetilde{g} : \widetilde{X} \times S \to \widetilde{Y} \times S$ for each strong equation $f \equiv g : X \to Y$ in Θ_{deco}.
- An equation $prl_Y \circ \widetilde{f} \equiv prl_Y \circ \widetilde{g} : \widetilde{X} \times S \to \widetilde{Y}$ for each weak equation $f \sim g : X \to Y$ in Θ_{deco}.
- A product $((\widetilde{q_i})_{(1)} : (\prod_j Y_j) \times S \to Y_i)_{i \in Loc}$ for each decorated product $(q_i^{(1)} : \prod_j Y_j \to Y_i)_{i \in Loc}$ in Θ_{deco}.

Thus, the morphism F_{expl} makes explicit the meaning of the decorations, by introducing a "type of states" S. In the following, the notation $\widetilde{\dots}$ will sometimes be omitted (mainly for types). The morphism F_{expl} is such that each modifier f gives rise to a term $\widetilde{f}$ which may use and modify the state, while whenever f is an accessor then $\widetilde{f}$ may use the state but is not allowed to modify it, and when moreover f is a pure term then $\widetilde{f}$ may neither use nor modify the state. When $f \equiv g$ then $\widetilde{f}$ and $\widetilde{g}$ must return the same result and the same state; when $f \sim g$ then $\widetilde{f}$ and $\widetilde{g}$ must return the same result but maybe not the same state.

Remark 3.3. When f and g are consecutive modifiers, we have defined $\widetilde{g \circ f} = \widetilde{g} \circ \widetilde{f}$. Thus, when f and g are accessors, the accessor $g \circ f$ is such that $\widetilde{g \circ f} = \langle \widetilde{g}_{(1)}, prr_Y \rangle \circ \widetilde{f} = \langle \widetilde{g}_{(1)} \circ \widetilde{f}, prr_Y \circ \widetilde{f} \rangle = \langle \widetilde{g}_{(1)} \circ \widetilde{f}, prr_X \rangle$, so that $\widetilde{g \circ f}_{(1)} = \widetilde{g}_{(1)} \circ \widetilde{f}$: we recognize the co-Kleisli composition of $\widetilde{f}_{(1)}$ and $\widetilde{g}_{(1)}$ with respect to the comonad $- \times S$. When f and g are pure then the pure term $g \circ f$ is such that $\widetilde{g \circ f}_{(0)} = \widetilde{g}_{(0)} \circ \widetilde{f}_{(0)}$.

Altogether, the span of logics for states $\mathcal{Z}_{\text{st}}$ is summarized in figure 5.

$\mathbf{T}_{\text{app}}$ $\overset{F_{\text{app}}}{\leftarrow}$	$\mathbf{T}_{\text{deco}}$ $\overset{F_{\text{expl}}}{\rightarrow}$	$\mathbf{T}_{\text{expl}}$
	modifier	
$f : X \to Y$	$f : X \to Y$	$\widetilde{f} : X \times S \to Y \times S$
	accessor	
$f : X \to Y$	$f^{(1)} : X \to Y$	$\widetilde{f}_{(1)} : X \times S \to Y$
	pure term	
$f : X \to Y$	$f^{(0)} : X \to Y$	$\widetilde{f}_{(0)} : X \to Y$
	strong equation	
$f \equiv g : X \to Y$	$f \equiv g : X \to Y$	$\widetilde{f} \equiv \widetilde{g} : X \times S \to Y \times S$
	weak equation	
$f \equiv g : X \to Y$	$f \sim g : X \to Y$	$prl_Y \circ \widetilde{f} \equiv prl_Y \circ \widetilde{g} : X \times S \to Y$

Figure 5: The span of logics for states

3.2 States as effect

Now let us introduce the operations and equations related to the states effect. We consider the semantics of states as the semantics of a language with effect, in the sense of definition 2.10, with respect to the span of logics for states $\mathcal{Z}_{\text{st}}$ defined in section 3.1. This language with effect $\Lambda_{\text{deco,st}} = (\Phi_{\text{deco,st}}, \Theta_{\text{deco,st}}, M_{\text{deco,st}})$ is defined below (the index "st" is omitted) in the following way:

- first the apparent syntax Φ_{app}, the decorated syntax Φ_{deco} and the explicit syntax $\Phi_{\text{expl}} = F_{\text{expl}}\Phi_{\text{deco}}$;

- then the explicit theory Θ_{expl} and the explicit semantics $M_{\text{expl}} : \Phi_{\text{expl}} \to \Theta_{\text{expl}}$, which form the expansion Λ_{expl} of Λ_{deco};
- and finally the decorated theory $\Theta_{\text{deco}} = G_{\text{expl}}\Theta_{\text{expl}}$ and the decorated semantics $M_{\text{deco}} = \varphi^{-1}M_{\text{expl}}$, where $\varphi : \text{Mod}_{\mathbf{T}}(\Phi, \Theta) \to \text{Mod}_{\mathbf{T}'}(\Phi', \Theta')$ is the bijection provided by the adjunction $F \dashv G$.

The apparent syntax Φ_{app} is built as follows. For each location i there is a type V_i for the possible values of i and an operation $l_i : \mathbb{1} \to V_i$ for observing the value of i. These operations form a product on *Loc* $(l_i : \mathbb{1} \to V_i)_{i \in Loc}$, so that for each location i there is an operation $u_i : V_i \to \mathbb{1}$, unique up to congruence, which satisfies the equations

$$\begin{cases} l_i \circ u_i \equiv id_{V_i} & : V_i \to V_i \\ l_j \circ u_i \equiv l_j \circ \langle\,\rangle_{V_i} & : V_i \to V_j \ \text{ for each } j \neq i \end{cases}$$

Intuitively, this means that after $u_i(a)$ is executed, the value of i is put to a and the value of j (for $j \neq i$) is unchanged.

Remark 3.4. Let Θ_{app} be the category of sets seen as a theory of the apparent logic (with equality as congruence). Let us try to build progressively a model of Φ_{app} in Θ_{app}. The type $\mathbb{1}$ must be interpreted as a singleton $\{*\}$, and for each i the interpretation of V_i is a set Val_i. Thus, the interpretation of l_i is an element of Val_i, and each interpretation of the V_i's and l_i's in Θ_{app} corresponds to a state, made of a value for each location; this is known as the *states-as-models* or *states-as-algebras* point of view [Gaudel et al. 1996]. This interpretation can be extended to $u_i : V_i \to \mathbb{1}$ in only one way: indeed u_i must be interpreted as the function which maps every value in Val_i to $*$. It follows that, as soon as the set Val_i is not a singleton, the equation $l_i \circ u_i \equiv id_{V_i}$ cannot be satisfied. Thus, the intended semantics of states cannot be a model of the apparent syntax Φ_{app} in Θ_{app}, as mentioned in remark 2.12.

The decorated syntax Φ_{deco} is obtained by adding informations (decorations) to Φ_{app}. It is generated by a type V_i and an accessor $l_i^{(1)} : \mathbb{1} \to V_i$ for each $i \in Loc$, which form a decorated product $(l_i^{(1)} : \mathbb{1} \to V_i)_{i \in Loc}$. The operations u_i's are decorated as modifiers and the equations as weak equations:

$$\begin{cases} l_i^{(1)} \circ u_i^{(2)} \sim id_{V_i}^{(0)} & : V_i \to V_i \\ l_j^{(1)} \circ u_i^{(2)} \sim l_j^{(1)} \circ \langle\,\rangle_{V_i}^{(0)} & : V_i \to V_j \ \text{ for each } j \neq i \end{cases} \tag{11}$$

It follows from the rules of the decorated logic that in every decorated theory there is an interpretation for the u_i's, which is unique up to strong equations. As required, the apparent syntax $\Phi_{\text{app}} = F_{\text{app}}\Phi_{\text{deco}}$ is recovered by dropping the decorations.

Using the definition of F_{expl} in section 3.1, we get the explicit syntax $\Phi_{\text{expl}} = F_{\text{expl}}\Phi_{\text{deco}}$. It is the theory in the explicit logic generated by a type V_i and a term $\widetilde{l}_{i(1)} : S \to V_i$ for each $i \in Loc$, which form a product $(\widetilde{l}_{i(1)} : S \to V_i)_{i \in Loc}$. So, for each location i, the operation $\widetilde{u}_i : V_i \times S \to S$ is defined up to congruence by the equations:

$$\begin{cases} \widetilde{l}_{i(1)} \circ \widetilde{u}_i \equiv prl_{V_i} & : V_i \times S \to V_i \\ \widetilde{l}_{j(1)} \circ \widetilde{u}_i \equiv \widetilde{l}_{j(1)} \circ prr_{V_i} & : V_i \times S \to V_j \ \text{ for each } j \neq i \end{cases}$$

The explicit theory Θ_{expl} is made of the category of sets with the equality as congruence, with a distinguished set *St* called the set of states, with cartesian products with *St*, and with a product on *Loc* with vertex *St*, denoted by $(l_i : St \to Val_i)_{i \in Loc}$, so that $St = \prod_{j \in Loc} Val_j$. The explicit semantics $M_{\text{expl}} : \Phi_{\text{expl}} \to \Theta_{\text{expl}}$ is the model (in the explicit logic) which maps S to *St* and, for each $i \in Loc$, the type V_i to the set Val_i and the operations l_i and u_i to the functions l_i and u_i, respectively.

The decorated semantics $M_{\text{deco}} : \Phi_{\text{deco}} \to \Theta_{\text{deco}}$ is obtained from the explicit semantics $M_{\text{expl}} : \Phi_{\text{expl}} \to \Theta_{\text{expl}}$ thanks to the adjunction $F_{\text{expl}} \dashv G_{\text{expl}}$. The decorated theory $\Theta_{\text{deco}} = G_{\text{expl}}\Theta_{\text{expl}}$ has a type for each set, a modifier $f^{(2)} : X \to Y$ for each function $f : X \times St \to Y \times St$, an accessor $f^{(1)} : X \to Y$ for each function $f : X \times St \to Y$ and a pure term $f^{(0)} : X \to Y$ for each function $f : X \to Y$, with the

straightforward conversions. It follows that there are in Θ_{deco}, for each $i \in Loc$, an accessor $l_i^{(1)} : \mathbb{1} \to Val_i$ and a modifier $u_i^{(2)} : Val_i \to \mathbb{1}$, and that we get the model $M_{\mathrm{deco}} = \varphi^{-1} M_{\mathrm{expl}}$ by mapping the type V_i to the set Val_i, the accessor $l_i^{(1)}$ to the function l_i and the modifier $u_i^{(2)}$ to the function u_i, for each $i \in Loc$.

According to claim 1.1, the explicit model M_{expl} provides the intended semantics of states. By adjunction, this is also the semantics of the decorated model M_{deco}, hence the following result.

Proposition 3.5. *The language with effect $\Lambda_{\mathrm{deco,st}}$ provides the intended semantics of states.*

To conclude this section, the decorated logic is used for proving a fundamental property of states: when a state s is modified by updating a location i with its own value in s, then the resulting state is undistinguishable from s; this is the first of equations 2. It should be reminded that each decorated proof may be mapped to an equational proof either by dropping the decorations (using the morphism F_{app}) or by expliciting them (using the morphism F_{expl}). In the first case one gets a correct proof which may be quite uninteresting, in the second case one gets a correct proof which may be quite complicated.

Proposition 3.6. *For every $i \in Loc$:*

$$\begin{cases} u_i^{(2)} \circ l_i^{(1)} \equiv id_{\mathbb{1}}^{(0)} & \textit{in the decorated logic} \\ \widetilde{u_i} \circ \widetilde{l_i} \equiv id_S & \textit{in the explicit logic} \end{cases}$$

Proof. In the decorated logic, let us prove the weak equations $l_j \circ u_i \circ l_i \sim l_j$ for each $j \in Loc$; then the first result will follow from the rule for decorated products on Loc and the second result by applying the morphism F_{expl}. In the following decorated proofs, the rules for associativity and identities are omitted. When $j = i$, the substitution rule for $\sim$ yields:

$$(\sim\text{-subs}) \ \frac{l_i \circ u_i \sim id_{V_i}}{l_i \circ u_i \circ l_i \sim l_i}$$

When $j \neq i$, using the substitution rule for $\sim$ and the replacement rule for $\equiv$ we get:

$$\dfrac{(\sim\text{-subs}) \ \dfrac{l_j \circ u_i \sim l_j \circ \langle\,\rangle_{V_i}}{l_j \circ u_i \circ l_i \sim l_j \circ \langle\,\rangle_{V_i} \circ l_i} \qquad (\equiv\text{-to-}\sim) \ \dfrac{(\equiv\text{-repl}) \ \dfrac{\dfrac{\vdots}{\langle\,\rangle_{V_i} \circ l_i \equiv id_{\mathbb{1}}}}{l_j \circ \langle\,\rangle_{V_i} \circ l_i \equiv l_j}}{l_j \circ \langle\,\rangle_{V_i} \circ l_i \sim l_j}}{l_j \circ u_i \circ l_i \sim l_j} \ (\sim\text{-trans})$$

□

4 Exceptions

It has been seen in section 1 that there is a duality between the semantics of states and the semantics of exceptions. A decorated language for states as effects has been designed in section 3. Now, in section 4.1, we define a decorated language for exceptions as effects simply by dualizing section 3; this provides the key operations for exceptions. Then in section 4.3 we check that the encapsulation of the key functions from section 1 may be performed in the decorated syntax.

4.1 Dualizing states

Let us dualize section 3. Let *ExCstr* be a given set, called the set of *exception constructors*. The span of logics for dealing with exceptions (with respect to *ExCstr*) is denoted by $\mathcal{Z}_{\mathrm{exc}}$:

In this section the subscript "exc" will be omitted. In order to focus on the fundamental properties of exceptions as effects, these logics are based on the *monadic equational logic*. Some additional features will be added in section 4.3.

A theory for the *decorated monadic equational logic for exceptions* $\mathbf{T}_{\text{deco}}$ is made of:

- Three nested monadic equational theories $\Theta^{(0)} \subseteq \Theta^{(1)} \subseteq \Theta^{(2)}$ with the same types, such that the congruence on $\Theta^{(0)}$ and on $\Theta^{(1)}$ is the restriction of the congruence $\equiv$ on $\Theta^{(2)}$. The objects of any of the three categories are called the *types* of the theory, the terms in $\Theta^{(2)}$ are the *catchers*, those in $\Theta^{(1)}$ are the *propagators* (or *throwers*) and those in $\Theta^{(0)}$ are the *pure terms*. The relations $f \equiv g$ are called the *strong equations*.
- An equivalence relation $\sim$ between parallel terms, which satisfies the properties of replacement and pure substitution (as in figure 6). The relations $f \sim g$ are called the *weak equations*. Every strong equation is a weak equation and every weak equation between propagators is a strong equation.
- A distinguished type $\mathbb{0}$ which has the following *decorated initiality* property: for each type X there is a pure term $[\,]_X : \mathbb{0} \to X$ such that every catcher $f : \mathbb{0} \to X$ satisfies $f \sim [\,]_X$.
- And Θ may have *decorated coproducts on ExCstr*, i.e., cocones of propagators $(q_i : X_i \to X)_{i \in ExCstr}$ such that for each cocone of propagators $(f_i : X_i \to Y)_{i \in ExCstr}$ with the same base there is a catcher $[f_j]_{j \in ExCstr} : X \to Y$ such that $\langle f_i \rangle_{i \in ExCstr} \circ q_i \sim f_i$ for each i, and whenever some catcher $g : X \to Y$ is such that $g \circ q_i \sim f_i$ for each i then $g \equiv [f_j]_{j \in ExCstr}$.

Figure 6 provides the *decorated rules* for exceptions, which describe the properties of the decorated theories. We use the following conventions: $X, Y, Z, \ldots$ are types, $f, g, h, \ldots$ are terms, $f^{(0)}$ means that f is a pure term, $f^{(1)}$ means that f is a propagator, and similarly $f^{(2)}$ means that f is a catcher (used for emphasizing). A decorated coproduct on *ExCstr* is denoted by $(q_i^{(1)} : X_i \to \sum_j X_j)_i$.

Remark 4.1. There is no general substitution rule for weak equations: if $f : X \to Y$ and $g_1 \sim g_2 : Y \to Z$ then in general $g_1 \circ f \not\sim g_2 \circ f$, except when f is pure.

On the "apparent" side of the span, a theory for the apparent logic $\mathbf{T}_{\text{app}}$ is a monadic equational theory with an initial object $\mathbb{0}$ which may have coproducts on *ExCstr*. The morphism $F_{\text{app}} : \mathbf{T}_{\text{deco}} \to \mathbf{T}_{\text{app}}$ maps each theory Θ_{deco} of $\mathbf{T}_{\text{deco}}$ to the theory Θ_{app} of $\mathbf{T}_{\text{app}}$ made of:

- A type $\widehat{X}$ for each type X in Θ_{deco}.
- A term $\widehat{f} : \widehat{X} \to \widehat{Y}$ for each catcher $f : X \to Y$ in Θ_{deco} (which includes the propagators and the pure terms), such that $\widehat{id_X} = id_{\widehat{X}}$ for each type X and $\widehat{g \circ f} = \widehat{g} \circ \widehat{f}$ for each pair of consecutive catchers (f, g).
- An equation $\widehat{f} \equiv \widehat{g}$ for each weak equation $f \sim g$ in Θ_{deco} (which includes the strong equations).
- A coproduct $(\widehat{q_i} : \widehat{X_i} \to \sum_j \widehat{X_j})_{i \in ExCstr}$ for each decorated coproduct $(q_i^{(1)} : X_i \to \prod_j X_j)_{i \in ExCstr}$ in Θ_{deco}.

Thus, the morphism F_{app} blurs the distinction between catchers, propagators and pure terms, and the distinction between weak and strong equations. In the following, the notation $\widehat{\ldots}$ will be omitted.

On the "explicit" side of the span, a theory for the explicit logic $\mathbf{T}_{\text{expl}}$ is a monadic equational theory with a distinguished object E, called the *type of exceptions*, with a coproduct-with-E functor $X + E$, and which may have coproducts on *ExCstr*. The morphism $F_{\text{expl}} : \mathbf{T}_{\text{deco}} \to \mathbf{T}_{\text{expl}}$ maps each theory $\Theta_{\text{deco}} = (\Theta^{(0)} \subseteq \Theta^{(1)} \subseteq \Theta^{(2)})$ of $\mathbf{T}_{\text{deco}}$ to the theory Θ_{expl} of $\mathbf{T}_{\text{expl}}$ made of:

- A type $\widetilde{X}$ for each type X in Θ_{deco}; the coprojections in $\widetilde{X} + E$ are denoted by $inl_X : \widetilde{X} \to \widetilde{X} + E$ and $inr_X : E \to \widetilde{X} + E$.

Rules of the monadic equational logic, and:

$$\frac{f^{(0)}}{f^{(1)}} \qquad \frac{f^{(1)}}{f^{(2)}}$$

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

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

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

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

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

$$\frac{(q_i^{(1)} : X_i \to \sum_j X_j)_i \;\; (f_i^{(1)} : X_i \to Y)_i}{[f_j]_j : X \to Y \;\; \forall i \, \langle f_i \rangle_i \circ q_i \sim f_i} \qquad \frac{(q_i^{(1)} : X_i \to \sum_j X_j)_i \;\; g : X \to Y \;\; \forall i \, g \circ q_i \sim f_i^{(1)}}{g \equiv [f_j]_j}$$

Figure 6: Rules of the decorated logic for exceptions

- A term $\widetilde{f} : \widetilde{X} + E \to \widetilde{Y} + E$ for each catcher $f : X \to Y$ in Θ_{deco}, such that:
  - if in addition f is a propagator then there is a term $\widetilde{f}_{(1)} : \widetilde{X} \to \widetilde{Y} + E$ such that $\widetilde{f} = [\widetilde{f}_{(1)} | inr_Y]$,
  - and if moreover f is a pure term then there is a term $\widetilde{f}_{(0)} : \widetilde{X} \to \widetilde{Y}$ such that $\widetilde{f}_{(1)} = inl_Y \circ \widetilde{f}_{(0)} : \widetilde{X} \to \widetilde{Y} + E$, hence $\widetilde{f} = [inl_Y \circ \widetilde{f}_{(0)} | inr_X] = \widetilde{f}_{(0)} + id_E$.

  and such that $\widetilde{id_X} = id_{\widetilde{X}+E}$ for each type X and $\widetilde{g \circ f} = \widetilde{g} \circ \widetilde{f}$ for each pair of consecutive catchers (f, g).
- An equation $\widetilde{f} \equiv \widetilde{g} : \widetilde{X} + E \to \widetilde{Y} + E$ for each strong equation $f \equiv g : X \to Y$ in Θ_{deco}.
- An equation $\widetilde{f} \circ inl_X \equiv \widetilde{g} \circ inl_Y : \widetilde{X} \to \widetilde{Y} + E$ for each weak equation $f \sim g : X \to Y$ in Θ_{deco}.
- A coproduct $((\widetilde{q_i})_{(1)} : (X_i \to (\sum_j X_j) + E)_{i \in ExCstr}$ for each decorated coproduct $(q_i^{(1)} : X_i \to \sum_j X_j)_{i \in ExCstr}$ in Θ_{deco}.

Thus, the morphism F_{expl} makes explicit the meaning of the decorations, by introducing a "type of exceptions" E which does not appear in the syntax. In the following, the notation $\widetilde{\ldots}$ will sometimes be omitted (mainly for types). The morphism F_{expl} is such that each catcher f gives rise to a term $\widetilde{f}$ which does not distinguish exceptions from ordinary values, while whenever f is a propagator then $\widetilde{f}$ may throw an exception but it must propagate exceptions, and when moreover f is a pure term then $\widetilde{f}$ must turn an ordinary value to an ordinary value and it must propagate exceptions. When $f \equiv g$ then $\widetilde{f}$ and $\widetilde{g}$ must coincide on ordinary

values and on exceptions; when $f \sim g$ then $\widetilde{f}$ and $\widetilde{g}$ must coincide on ordinary values but maybe not on exceptions.

Remark 4.2. When f and g are consecutive catchers, we have defined $\widetilde{g \circ f} = \widetilde{g} \circ \widetilde{f}$. Thus, dually to remark 3.3, when f and g are propagators then the propagator $g \circ f$ is such that $\widetilde{g \circ f}_{(1)}$ is the Kleisli composition of $\widetilde{f}_{(1)}$ and $\widetilde{g}_{(1)}$ with respect to the monad $- + E$, and when f and g are pure then the pure term $g \circ f$ is such that $\widetilde{g \circ f}_{(0)} = \widetilde{g}_{(0)} \circ \widetilde{f}_{(0)}$.

Altogether, the span of logics for exceptions $\mathcal{Z}_{\text{exc}}$ is summarized in figure 7.

$\mathbf{T}_{\text{app}}$	$\overset{F_{\text{app}}}{\leftarrow}$	$\mathbf{T}_{\text{deco}}$	$\overset{F_{\text{expl}}}{\rightarrow}$	$\mathbf{T}_{\text{expl}}$
		catcher		
$f : X \to Y$		$f : X \to Y$		$\widetilde{f} : X + E \to Y + E$
		propagator		
$f : X \to Y$		$f^{(1)} : X \to Y$		$\widetilde{f}_{(1)} : X \to Y + E$
		pure term		
$f : X \to Y$		$f^{(0)} : X \to Y$		$\widetilde{f}_{(0)} : X \to Y$
		strong equation		
$f \equiv g : X \to Y$		$f \equiv g : X \to Y$		$\widetilde{f} \equiv \widetilde{g} : X + E \to Y + E$
		weak equation		
$f \equiv g : X \to Y$		$f \sim g : X \to Y$		$\widetilde{f} \circ \mathit{inl}_X \equiv \widetilde{g} \circ \mathit{inl}_X : X \to Y + E$

Figure 7: The span of logics for exceptions

Now we consider the semantics of exceptions as the semantics of a language with effect $\Lambda_{\text{deco,exc}} = (\Phi_{\text{deco,exc}}, \Theta_{\text{deco,exc}}, M_{\text{deco,exc}})$ with respect to the span of logics $\mathcal{Z}_{\text{exc}}$.

The apparent syntax Φ_{app} is built as follows. For each exception constructor i there is a type P_i for the possible parameters and an operation $t_i : P_i \to \mathbb{0}$ called the *key thrower*, for throwing an exception of constructor i. These operations form a coproduct on ExCstr $(t_i : P_i \to \mathbb{0})_{i \in \mathit{ExCstr}}$, so that for each i there is an operation $c_i : \mathbb{0} \to P_i$, unique up to congruence), called the *key catcher*, which satisfies the equations

$$\begin{cases} c_i \circ t_i \equiv id_{P_i} & : P_i \to P_i \\ c_i \circ t_j \equiv [\,]_{P_i} \circ t_j & : P_j \to P_i \ \text{ for each } j \neq i \end{cases}$$

Intuitively, this means that when c_i is called, the parameter of the previous call to t_i (for the same i) is returned.

The decorated syntax Φ_{deco} is obtained by adding informations (decorations) to Φ_{app}. It is generated by a type P_i and a propagator $t_i^{(1)} : P_i \to \mathbb{0}$ for each $i \in \mathit{ExCstr}$, which form a decorated coproduct $(t_i^{(1)} : P_i \to \mathbb{0})_{i \in \mathit{ExCstr}}$. The operations c_i's are decorated as catchers and the equations as weak equations:

$$\begin{cases} c_i^{(2)} \circ t_i^{(1)} \sim id_{P_i}^{(0)} & : P_i \to P_i \\ c_i^{(2)} \circ t_j^{(1)} \sim [\,]_{P_i}^{(0)} \circ t_j^{(1)} & : P_j \to P_i \ \text{ for each } j \neq i \end{cases} \tag{12}$$

It follows from the rules of the decorated logic that in every decorated theory there is an interpretation for the c_i's, which is unique up to strong equations. The apparent syntax $\Phi_{\text{app}} = F_{\text{app}}\Phi_{\text{deco}}$ is recovered by dropping the decorations. The explicit syntax $\Phi_{\text{expl}} = F_{\text{expl}}\Phi_{\text{deco}}$ is the theory in the explicit logic generated

by a type P_i and a term $\widetilde{t}_{i(1)} : P_i \to E$ for each $i \in ExCstr$, which form a coproduct $(\widetilde{t}_{i(1)} : P_i \to E)_{i \in ExCstr}$. So, for each i, the operation $\widetilde{c}_i : E \to P_i + E$ is defined up to strong equations by the weak equations:

$$\begin{cases} \widetilde{c}_i \circ \widetilde{t}_{i(1)} \sim inl_{P_i} & : P_i \to P_i + E \\ \widetilde{c}_i \circ \widetilde{t}_{j(1)} \sim inr_{P_i} \circ \widetilde{t}_{j(1)} & : P_j \to P_i + E \ \text{ for each } j \neq i \end{cases}$$

The explicit theory Θ_{expl} is made of the category of sets with the equality as congruence, with a distinguished set Exc called the set of exceptions, with disjoint unions with Exc, and with a coproduct on $ExCstr$ with vertex Exc, denoted by $(t_i : Par_i \to Exc)_{i \in ExCstr}$, so that $Exc = \sum_{j \in ExCstr} Par_j$. The explicit semantics $M_{\text{expl}} : \Phi_{\text{expl}} \to \Theta_{\text{expl}}$ is the model (in the explicit logic) which maps E to Exc and, for each $i \in ExCstr$, the type P_i to the set Par_i and the operations t_i and c_i to the functions t_i and c_i, respectively. The decorated semantics $M_{\text{deco}} : \Phi_{\text{deco}} \to \Theta_{\text{deco}}$ is obtained from the explicit semantics $M_{\text{expl}} : \Phi_{\text{expl}} \to \Theta_{\text{expl}}$ thanks to the adjunction $F_{\text{expl}} \dashv G_{\text{expl}}$.

The next result is dual to proposition 3.6, it can be proved in the dual way, using the decorated logic for exceptions. It is the key lemma for proving proposition 4.8, which says that catching an exception of constructor i by throwing the same exception is like doing nothing.

Proposition 4.3. *For every $i \in ExCstr$:*

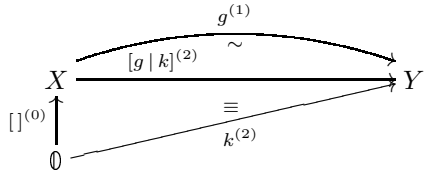
$$\begin{cases} t_i^{(1)} \circ c_i^{(2)} \equiv id_{\mathbb{0}}^{(0)} & \textit{in the decorated logic} \\ \widetilde{t}_i \circ \widetilde{c}_i \equiv id_E & \textit{in the explicit logic} \end{cases}$$

4.2 Extending the decorated logic

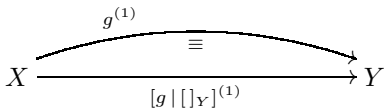
In the previous section 4.1 the key operations $t_i^{(1)}$'s and $c_i^{(2)}$'s have been defined; in the next section 4.3 they will be used for building the decorated raising and handling operations. For this purpose, some rules must be added to the decorated logic for exceptions; this is done now.

Definition 4.4. The decorated logic for exceptions $\mathbf{T}_{\text{deco,exc}}$ is extended as $\mathbf{T}^{+}_{\text{deco,exc}}$ by adding the following rules.

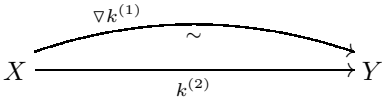
- For each point X there is a *decorated sum* $X = X + \mathbb{0}$, in the sense that for each propagator $g^{(1)} : X \to Y$ and each catcher $k^{(2)} : \mathbb{0} \to Y$ there is a catcher $[g \mid k]^{(2)} : X \to Y$, unique up to strong equations, such that $[g \mid k]^{(2)} \sim g^{(1)}$ and $[g \mid k]^{(2)} \circ [\,]_X^{(0)} \equiv k^{(2)}$.



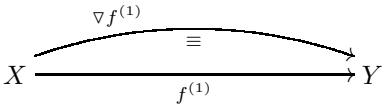
  In addition, whenever $f^{(1)} : \mathbb{0} \to Y$ is a propagator (which implies that $f^{(1)} \equiv [\,]_Y^{(0)}$) then $[g \mid f] : X \to Y$ is a propagator (so that the weak equation $[g \mid f]^{(1)} \sim g^{(1)}$ is strong: $[g \mid f]^{(1)} \equiv g^{(1)}$). In particular, we will use the fact that $[g^{(1)} \mid [\,]_Y^{(0)}] \equiv g^{(1)}$.



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



  Thus, whenever $f^{(1)} : X \to Y$ is a propagator then $\triangledown f^{(1)} \equiv f^{(1)}$.



Let us check that $\mathbf{T}_{\text{deco,exc}}$ can be replaced by $\mathbf{T}^{+}_{\text{deco,exc}}$ in the span of logics $\mathcal{Z}_{\text{deco}}$: we have to check that both morphisms $F_{\text{app,deco}}$ and $F_{\text{expl,deco}}$ map the new rules to rules in the apparent and in the explicit logic, respectively. This is obvious for the apparent logic, where $k \equiv [\,]_Y$, $[g \,|\, k] \equiv g$ and $\triangledown f \equiv f$. For the explicit logic, for each $\widetilde{g}_{(1)} : X \to Y + E$ and $\widetilde{k} : E \to Y + E$ we have $\widetilde{[g \,|\, k]} = [\widetilde{g}_{(1)} \,|\, \widetilde{k}] : X + E \to Y + E$. It follows that $\widetilde{[g \,|\, [\,]_Y]} \equiv \widetilde{g}$, which propagates exceptions. And for each $\widetilde{k} : X + E \to Y + E$ we have $\widetilde{\triangledown k}_{(1)} \equiv \widetilde{k} \circ inl_X : X \to Y + E$.

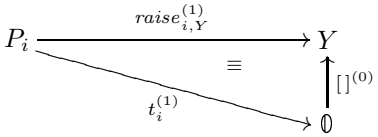
4.3 Encapsulating exceptions

This section is a "decorated version" of section 1.2. We show that the $t_i^{(1)}$'s and $c_i^{(2)}$'s are the key operations for dealing with exceptions in the decorated logic. More precisely, we prove that, for each constructor i, the *raise* operation is built from pure operations and a unique propagator t_i, and the *handle* operation is built from propagators and a unique catcher c_i. There are at least two reasons for not using t_i and c_i directly: firstly in a programming language there is usually no name and no intuition for the "empty" type $\mathbb{0}$, secondly the handling of exceptions is a powerful programming technique which must be carefully encapsulated: while most operations are allowed to throw exceptions, only some very special operations are allowed to catch exceptions.

First, let us focus on *raising* exceptions. This operation is a propagator, it calls the key thrower $t_i^{(1)} : P_i \to \mathbb{0}$ and "hides" the empty type by mapping it into the required type of results.

Definition 4.5. For each i in *ExCstr* and each object Y, the propagator "*raise (or throw) an exception of constructor i in Y*" is

$$raise_{i,Y}^{(1)} = throw_{i,Y}^{(1)} = [\,]_Y^{(0)} \circ t_i^{(1)} : P_i \to Y$$



(13)

Now, let us consider the *handling* of exceptions, which calls the key catchers $c_i^{(2)}$'s. Let $f^{(1)} : X \to Y$ be some propagator. For handling exceptions of constructors $i_1, \ldots, i_n$ raised by f, using propagators $g_1^{(1)} : Par_{i_1} \to Y, \ldots, g_n^{(1)} : Par_{i_n} \to Y$, the handling process builds a propagator:

$$(f \; handle \; i_1 \Rightarrow g_1 \,|\, \ldots \,|\, i_n \Rightarrow g_n)^{(1)} \;=\; (try\{f\} \; catch \, i_1 \, \{g_1\} \; catch \, i_2 \, \{g_2\} \ldots catch \, i_n \, \{g_n\})^{(1)}$$

which is also denoted in a more compact way as

$$(f \; handle \; (i_k \Rightarrow g_k)_{1 \le k \le n})^{(1)} \;=\; (try\{f\} \; catch \, i_k \{g_k\}_{1 \le k \le n})^{(1)} : X \to Y$$

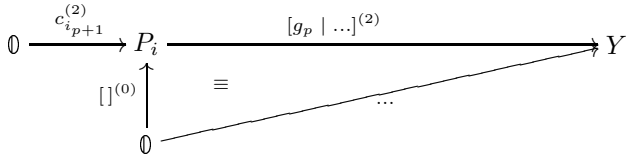
Definition 4.6. For each propagator $f^{(1)} : X \to Y$, each $n \geq 1$, each exception constructors $i_1, ..., i_n$ and each propagators $g_1^{(1)} : P_{i_1} \to Y, ..., g_n^{(1)} : P_{i_n} \to Y$, the propagator "*handle the exception e raised in f, if any, with g_1 if e has constructor i_1, otherwise with g_2 if e has constructor i_2, ..., otherwise with g_n if e has constructor i_n*", is

$$(f \ handle \ (i_k \Rightarrow g_k)_{1 \leq k \leq n})^{(1)} \ = \ (try\{f\} \ catch \ i_k\{g_k\}_{1 \leq k \leq n})^{(1)} : X \to Y$$

defined as follows.

1. The catchers $(catch \ i_k\{g_k\}_{p \leq k \leq n})^{(2)} : \mathbb{0} \to Y$ are defined recursively by

$$(catch \ i_k\{g_k\}_{p \leq k \leq n})^{(2)} \ = \ \begin{cases} [g_p^{(1)} \mid (catch \ i_k\{g_k\}_{p+1 \leq k \leq n})^{(2)}]^{(1)} \circ c_{i_p}^{(2)} & \text{when } p < n \\ g_n^{(1)} \circ c_{i_n}^{(2)} & \text{when } p = n \end{cases}$$

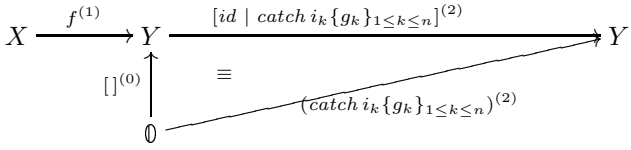


(14)

   where ... stands for $(catch \ i_k\{g_k\}_{p+1 \leq k \leq n})^{(2)}$ when $p < n$ and for $[\,]_Y^{(0)}$ when $p = n$, since $[g_n^{(1)} \mid [\,]_Y^{(0)}]^{(1)} \equiv g_n^{(1)}$.

2. Then the catcher $H^{(2)} : X \to Y$ is defined as

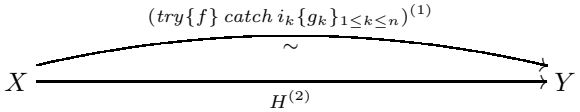
$$H^{(2)} \ = \ [id_Y^{(0)} \mid (catch \ i_k\{g_k\}_{1 \leq k \leq n})^{(2)}] \circ f^{(1)} : X \to Y$$



(15)

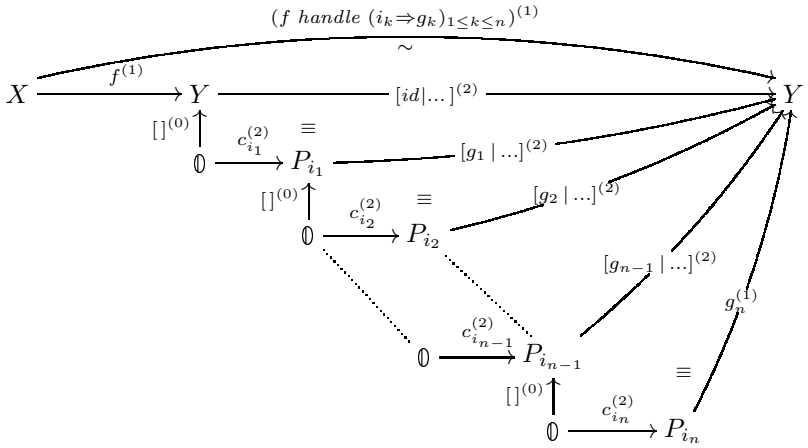
3. Finally the handling function is the propagator $(\triangledown H)^{(1)}$

$$(try\{f\} \ catch \ i_k\{g_k\}_{1 \leq k \leq n})^{(1)} \ = \ (\triangledown H)^{(1)} : X \to Y$$



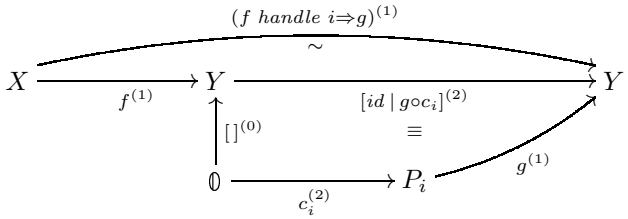
(16)

Altogether, we get:



When $n = 1$, this becomes simply:

$$(f \ \mathit{handle}\ i \Rightarrow g)^{(1)} = (\mathit{try}\{f\}\ \mathit{catch}\, i\, \{g\})^{(1)} = \triangledown([id_Y \mid g \circ c_i] \circ f) : X \to Y$$



It is easy to check that by applying the expansion morphism F_{expl} to the decorated definitions of *raise* and *handle* we get the explicit description from section 1.2: diagrams 3, 4, 5, 6, are mapped respectively to diagrams 13, 14, 15, 16. According to claim 1.5, the explicit language $\Lambda_{\text{expl,exc}}$ provides the intended semantics of exceptions. By adjunction (remark 2.11) this is also the semantics of the language with effect $\Lambda_{\text{deco,exc}}$, hence the following result.

Proposition 4.7. *The language with effect $\Lambda_{\text{deco,exc}}$ provides the intended semantics of exceptions.*

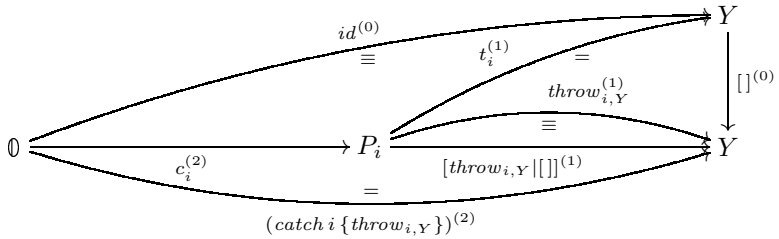
4.4 Some properties of exceptions

The next proposition shows that catching an exception of constructor i by throwing the same exception is like doing nothing. Indeed, by expansion this result implies that in the semantics (as in section 1.2) for all $e \in \mathit{Exc}$, $(\mathit{catch}\, i\, \{\mathit{throw}_{i,Y}\})(e) = e \in \mathit{Exc} \subseteq Y + \mathit{Exc}$.

Proposition 4.8. *For every $i \in \mathit{ExCstr}$, in the decorated logic:*

$$(\mathit{catch}\, i\, \{\mathit{throw}_{i,Y}\})^{(2)} \equiv [\,]_Y^{(0)}$$

Proof. Let $g = \mathit{throw}_{i,Y} : P_i \to Y$. By definition 4.6 we have $\mathit{catch}\, i\, \{g\} = [g|[\,]_Y] \circ c_i : 0 \to Y$, and since g is a propagator, by definition 4.4 we have $[g|[\,]_Y] \equiv g$, so that $\mathit{catch}\, i\, \{g\} \equiv g \circ c_i$. By definition 4.5 we have $\mathit{throw}_{i,Y} = [\,]_Y \circ t_i : P_i \to Y$ hence $\mathit{catch}\, i\, \{g\} \equiv [\,]_Y \circ t_i \circ c_i$, and since $t_i \circ c_i \equiv id_0$ by proposition 4.3 we get $\mathit{catch}\, i\, \{\mathit{throw}_{i,Y}\} \equiv [\,]_Y$.



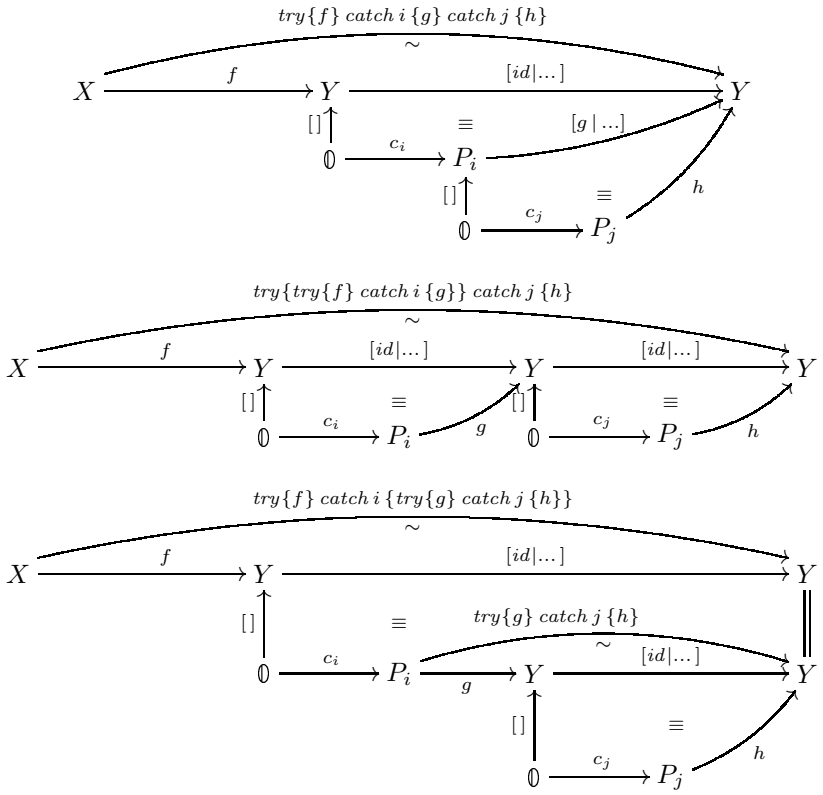
□

Remark 4.9. The three propagators

$$\begin{cases} \mathit{try}\{f\}\ \mathit{catch}\, i\, \{g\}\ \mathit{catch}\, j\, \{h\} \\ \mathit{try}\{\mathit{try}\{f\}\ \mathit{catch}\, i\, \{g\}\}\ \mathit{catch}\, j\, \{h\} \\ \mathit{try}\{f\}\ \mathit{catch}\, i\, \{\mathit{try}\{g\}\ \mathit{catch}\, j\, \{h\}\} \end{cases}$$

do not behave in the same way: whenever $f(x)$ raises an exception $t_i(a)$ of constructor i and $g(a)$ raises an exception $t_j(b)$ of constructor j, the first propagator returns $t_j(b)$ (uncaught) while the second and the third ones return $h(b)$; whenever $f(x)$ raises an exception $t_j(b)$ of constructor j, the first and the second

propagators return $h(b)$ while the third one returns $t_j(b)$ (uncaught). The differences can be seen from the diagrams.



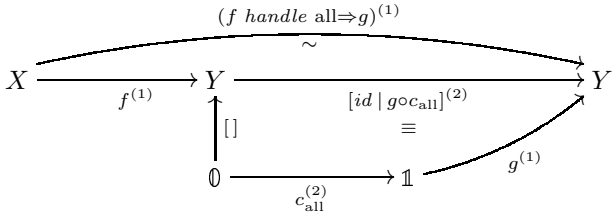
The next result is proved in appendix A.2.

Proposition 4.10. *For every $i, j \in ExCstr$, in the decorated logic:*

$$try\{f\}\ catch\, i\,\{g\}\ catch\, j\,\{h\} \equiv try\{f\}\ catch\, j\,\{h\}\ catch\, i\,\{g\}\ \text{ if } i \neq j$$
$$try\{f\}\ catch\, i\,\{g\}\ catch\, i\,\{h\} \equiv try\{f\}\ catch\, i\,\{g\}$$

Remark 4.11. The *catch* construction is easily extended to a *catch-all* construction like `catch(...)` in `C++`. We add to the decorated logic for exceptions a pure unit type $\mathbb{1}$, which means, a type $\mathbb{1}$ such that for each type X there is a pure term $\langle\rangle_X : X \to \mathbb{1}$, unique up to strong equations. Then we add a catcher $c^{(2)}_{\text{all}} : \mathbb{0} \to \mathbb{1}$ with the equations $c_{\text{all}} \circ t_j \sim \langle\rangle_{P_j}$ for every $j \in ExCstr$, which means that c_{all} catches exceptions of the form $t_j(a)$ for every j and forgets the value a. For each propagators $f^{(1)} : X \to Y$ and $g^{(1)} : \mathbb{1} \to Y$, the propagator "*handle the exception e raised in f, if any, with g*" is

$$(f\ handle\ \text{all} \Rightarrow g)^{(1)} = \triangledown([id_Y \mid g \circ c_{\text{all}}] \circ f) : X \to Y$$



The semantics of the *catch-all* construction is easily derived from this diagram, as a function $(f\ handle\ \text{all} \Rightarrow g) : X \to Y + Exc$ where $Exc = \sum_{j \in ExCstr} Par_j$ and $g : \mathbb{1} \to Y + Exc$ is a constant:

For each $x \in X + Exc$, $(f \ handle \ \text{all} \Rightarrow g)(x) \in Y + Exc$ is defined by:
if $x \in Exc$ then return $x \in Exc \subseteq Y + Exc$;
// *now x is not an exception*
compute $y := f(x) \in Y + Exc$;
if $y \in Y$ then return $y \in Y \subseteq Y + Exc$;
// *now y is an exception*
return $g \in Y + Exc$.

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

5 The duality

The previous results are summarized in section 5.1, then some remarks about other semantical issues are outlined in section 5.2.

5.1 Duality of states and exceptions as effects

Given a set I, let $\mathcal{Z}_{\text{deco,st}}$ be the span of diagrammatic logics for states with respect to the set of locations I as defined in section 3.1. In this span, let $\Lambda_{\text{deco,st}}$ be the language with effects for states as defined in section 3.2. Then proposition 3.5 states that $\Lambda_{\text{deco,st}}$ provides the intended semantics of states.

Given a set I, let $\mathcal{Z}_{\text{deco,exc}}$ be the span of diagrammatic logics for exceptions with respect to the set of exceptions constructors I as defined in section 4.1. In this span, let $\Lambda_{\text{deco,exc}}$ be the language with effects for exceptions as defined in section 4.1. Then proposition 4.7 states that $\Lambda_{\text{deco,exc}}$ provides the intended semantics of exceptions. It should be reminded that the whole process of raising and handling exceptions does rely on the key functions t_i and c_i: this has been checked in section 4.3.

Figure 8 recapitulates the properties of the functions *lookup* (l_i) and *update* (u_i) for states on the left, and the properties of the functions *key throw* (t_i) and *key catch* (c_i) for exceptions on the right. By expansion, figure 8 gives rise to figure 1. Our main result (theorem 5.1) follows immediately; it means that the well-known duality between categorical products and coproducts can be extended as a duality between the lookup and update functions for states on one side and the key throwing and catching functions for exceptions on the other. The notion of opposite categories and duality is extended in the straightforward way to diagrammatic logics and to spans of diagrammatic logics.

Theorem 5.1. *With the previous notations, the span of diagrammatic logics $\mathcal{Z}_{\text{exc}}$ for exceptions is opposite to the span of diagrammatic logics $\mathcal{Z}_{\text{st}}$ for states and the language with effects $\Theta_{\text{deco,exc}}$ for exceptions is dual to the language with effects $\Theta_{\text{deco,st}}$ for states.*

5.2 Other semantics

Equations (12) relating the key throw and catch operations may be oriented from left to right in order to get the usual *operational semantics* of exceptions: when an exception is thrown by some occurrence of t_i, the execution jumps to the first occurrence of c_i and wipes out the pair (t_i, c_i) and everything between them.

In a dual way, equations (11) relating the lookup and update operations may be oriented from left to right, but this does not provide the usual operational semantics of states. In fact, equations (11) are related to the *Hoare-Floyd semantics* of states: they give rise to the basic occurrences of the assignment axiom $\{G[e/X]\}\ X := e\ \{G\}$, namely:

$$\{e = n\}\ X := e\ \{X = n\} \ \text{ and } \ \{Y = n\}\ X := e\ \{Y = n\} \ \text{ when } \ Y \neq X$$

¿From the decorated point of view, the value n is pure, the expressions e, $e = n$, $X = n$ and $Y = n$ are accessors, the command $X := e$ is a modifier and the equalities are weak equations. The axioms mean that

States	Exceptions
$i \in Loc,\ Val_i,$	$i \in ExCstr,\ Par_i,$
$\mathbb{1}$ terminal	$\mathbb{0}$ initial
$\langle\,\rangle_i^{(0)} : Val_i \to \mathbb{1}$	$\mathbb{0} \leftarrow Par_i : [\,]_i^{(0)}$
$l_i^{(1)} : \mathbb{1} \to Val_i$	$\mathbb{0} \leftarrow Par_i : t_i^{(1)}$
$u_i^{(2)} : Val_i \to \mathbb{1}$	$Par_i \leftarrow \mathbb{0} : c_i^{(2)}$
$\begin{array}{ccc} Val_i & \xrightarrow{id} & Val_i \\ {\scriptstyle u_i}\downarrow & \underset{l_i}{\sim} & \downarrow{\scriptstyle id} \\ \mathbb{1} & \xrightarrow{\quad} & Val_i \end{array}$ $\begin{array}{ccccc} Val_i & \xrightarrow{\langle\,\rangle} & \mathbb{1} & \xrightarrow{l_j} & Val_j \\ {\scriptstyle u_i}\downarrow & & \underset{l_j}{\sim} & & \downarrow{\scriptstyle id} \\ \mathbb{1} & & \xrightarrow{\quad} & & Val_j \end{array}$ $(j \neq i)$	$\begin{array}{ccc} Par_i & \xleftarrow{id} & Par_i \\ {\scriptstyle c_i}\uparrow & \underset{t_i}{\sim} & \uparrow{\scriptstyle id} \\ \mathbb{0} & \xleftarrow{\quad} & Par_i \end{array}$ $\begin{array}{ccccc} Par_i & \xleftarrow{[\,]} & \mathbb{0} & \xleftarrow{t_j} & Par_j \\ {\scriptstyle c_i}\uparrow & & \underset{t_j}{\sim} & & \uparrow{\scriptstyle id} \\ \mathbb{0} & & \xleftarrow{\quad} & & Par_j \end{array}$ $(j \neq i)$

Figure 8: Duality of decorated syntax

whenever $e^{(1)} \sim n^{(0)}$ then $l_X^{(1)} \circ u_X^{(2)} \circ e^{(1)} \sim n^{(0)}$ and $l_Y^{(1)} \circ u_X^{(2)} \circ e^{(1)} \sim l_Y^{(1)}$ if $Y \neq X$, which is easily derived from equations 11.

Conclusion

We have discovered a symmetry between the key notions underlying the effects of states and exceptions, thanks to our approach of computational effects relying on spans of diagrammatic logics. A consequence is that the duality principle can be applied for deriving properties of exceptions from the properties of states. Another consequence is that this symmetry provides a new point of view on exceptions, mainly by distinguishing the key catching operation from the surrounding conditionals in the handling process.

This symmetry between states and exceptions is deeply hidden, which may explain that our result is, as far as we know, completely new. First, as seen in the paper, for states the key operations are visible, while for exceptions they are encapsulated. In addition, most features which we might want to add will contribute to hide the duality: this happens for instance simply when adding pure constants $a^{(0)} : \mathbb{1} \to V_i$ for states and $a^{(0)} : \mathbb{1} \to P_i$ for exceptions, not $a^{(0)} : P_i \to \mathbb{0}$. Adding products on one side and coproducts on the other, as in appendix A, preserves the duality. Adding both products and coproducts on either side preserves the duality, but the distributivity or extensivity property, which is usually assumed, does not preserve it. Adding exponentials in order to get a lambda-calculus would be desirable, but this might further obscure the duality. Many questions are still open, for instance about a similar duality applying for other effects, or about the combination of effects from this point of view.

References

[Adámek et al. 2011] Jiří Adámek, Jiří Rosický, Enrico M. Vitale. Algebraic Theories: A Categorical Introduction to General Algebra. Cambridge University Press (2011).

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

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

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

[Duval & Reynaud 2005] Dominique Duval, Jean-Claude Reynaud. Diagrammatic logic and exceptions: an introduction. Mathematics, Algorithms, Proofs. Dagstuhl Seminar Proceedings 05021 (2006).

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

[Ehrig & Mahr 1985] Hartmut Ehrig, Bernd Mahr. Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. Springer-Verlag Monographs in Theoretical Computer Science 6 (1985).

[Filinski 1994] Andrzej Filinski. Representing Monads. POPL 1994. ACM Press, p. 446-457 (1994).

[Gabriel & Ulmer 1971] Peter Gabriel, Friedrich Ulmer. Lokal präsentierbare Kategorien. Springer-Verlag Lecture Notes in Mathematics 221 (1971).

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

[Gaudel et al. 1996] Marie-Claude Gaudel, Pierre Dauchy, Carole Khoury. A Formal Specification of the Steam-Boiler Control Problem by Algebraic Specifications with Implicit State. Formal Methods for Industrial Applications 1995. Springer-Verlag Lecture Notes in Computer Science 1165, p. 233-264 (1996).

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

[Hyland & Power 2007] 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).

[Java] The Java Language Specification, Third Edition. `http://java.sun.com/docs/books/jls/third_edition/html/sta`

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

[Makkai 1997] Michael Makkai. Generalized sketches as a framework for completeness theorems (I). Journal of Pure and Applied Algebra 115, p. 49-79 (1997).

[Melliès 2010] Paul-André Melliès. Segal condition meets computational effects. LICS 2010. IEEE Computer Society, p. 150-159 (2010).

[Moggi 1989] Eugenio Moggi. Computational Lambda-Calculus and Monads. LICS 1989 IEEE Conference Proceedings, p. 14-23 (1989).

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

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

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

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

[Schröder & Mossakowski 2004] 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).

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

A

In this appendix we consider two equations for states: equations (6) and (3) in the list of equations 2 (section 1.1), and the dual equations for exceptions, all of them in the decorated logic. In the decorated proofs below, the associativity and identity rules are skipped and the decoration of morphisms is often omitted.

A.1 States

Equations (6) and (3) in the list 2 (section 1.1) are:

$$(6)\quad \forall i \neq j \in Loc,\ \forall\, s \in St,\ a \in Val_i,\ b \in Val_j,\ \ u_j(b, u_i(a,s)) = u_i(a, u_j(b,s)) \in St$$

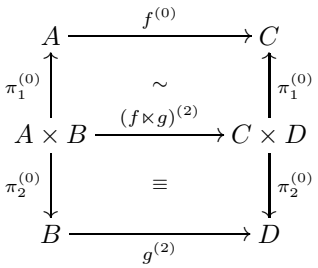
$$(3)\quad \forall i \in Loc,\ \forall\, s \in St,\ a, a' \in Val_i,\ \ u_i(a', u_i(a,s)) = u_i(a',s) \in St$$

Since these equations have two values as arguments, we will use the notion of semi-pure product from [Dumas et al. 2011]. In the decorated logic for states, the *product* of two objects A and B is an object $A \times B$ with two pure morphisms: the projections $\pi_1^{(0)} : A \times B \to A$ and $\pi_2^{(0)} : A \times B \to B$, which satisfy the usual categorical product property with respect to the pure morphisms (so that, as usual, the projections $\pi_1^{(0)} : A \times \mathbb{1} \to A$ and $\pi_2^{(0)} : \mathbb{1} \times B \to B$ are isomorphisms). The *product* of two pure morphisms $f^{(0)} : A \to C$ and $g^{(0)} : B \to D$ is a pure morphism $(f \times g)^{(0)} : A \times B \to C \times D$ which is characterized, up to $\equiv$, by:

$$\pi_1^{(0)} \circ (f \times g)^{(0)} \equiv f^{(0)} \circ \pi_1^{(0)}$$
$$\pi_2^{(0)} \circ (f \times g)^{(0)} \equiv g^{(0)} \circ \pi_2^{(0)}$$

Such a property, symmetric in f and g, cannot be satisfied by modifiers: indeed, the effect of building a pair of modifiers depends on the evaluation strategy. However, in [Dumas et al. 2011] we define the *semi-pure product* of a pure morphism $f^{(0)} : A \to C$ and a modifier $g^{(2)} : B \to D$, as a modifier $(f \ltimes g)^{(2)} : A \times B \to C \times D$ which is characterized, up to $\equiv$, by the following decorated version of the product property:

$$\pi_1^{(0)} \circ (f \ltimes g)^{(2)} \sim f^{(0)} \circ \pi_1^{(0)} \qquad \text{P1}$$
$$\pi_2^{(0)} \circ (f \ltimes g)^{(2)} \equiv g^{(2)} \circ \pi_2^{(0)} \qquad \text{P2}$$



The weak equations 11 relating the functions $(u_i^{(2)})_{i\in Loc}$ and $(l_i^{(1)})_{i\in Loc}$ will be used as axioms in the proof trees with the following labels:

$$l_i^{(1)} \circ u_i^{(2)} \sim id_{V_i}^{(0)} : V_i \to V_i \qquad \text{A1}$$

$$l_j^{(1)} \circ u_i^{(2)} \sim l_j^{(1)} \circ \langle\,\rangle_{V_i}^{(0)} : V_i \to V_j \ \text{ for each } j \neq i \qquad \text{A2}$$

Equation (6) is expressed in the decorated logic as:

$$(6)_{\text{st}}\ \forall i \neq j \in Loc,\ u_j^{(2)} \circ \pi_2^{(0)} \circ (u_i \rtimes id_{V_j})^{(2)} \equiv u_i^{(2)} \circ \pi_1^{(0)} \circ (id_{V_i} \ltimes u_j)^{(2)}$$

This strong equation is equivalent to the family of weak equations:

$$(6)_{\text{st,obs}}\ \forall k \in Loc, \forall i \neq j \in Loc,\ l_k^{(1)} \circ u_j^{(2)} \circ \pi_2^{(0)} \circ (u_i \rtimes id_{V_j})^{(2)} \sim l_k^{(1)} \circ u_i^{(2)} \circ \pi_1^{(0)} \circ (id_{V_i} \ltimes u_j)^{(2)}$$

So, let $i, j, k \in Loc$ with $i \neq j$.

1. For $k \neq i, j$, the weak equation

   $$l_k \circ u_j \circ \pi_2 \circ (u_i \rtimes id_{V_j}) \sim l_k \circ \langle\,\rangle_{V_i \times V_j}.$$

   is proven in Figure 9 (proof Pr_4). A symmetric proof shows that

   $$l_k \circ u_i \circ \pi_1 \circ (id_{V_i} \ltimes u_j) \sim l_k \circ \langle\,\rangle_{V_i \times V_j}$$

   With the symmetry and transitivity of $\sim$ this concludes the proof of equations $(6)_{\text{st,obs}}$ when $k \neq i, j$.

2. When $k = i$, the weak equations

   $$l_i \circ u_j \circ \pi_2 \circ (u_i \rtimes id_{V_j}) \sim \pi_1$$
   $$l_i \circ u_i \circ \pi_1 \circ (id_{V_i} \ltimes u_j) \sim \pi_1$$

   are proven in Figure 10 (proofs Pr_7 and Pr_8). With the symmetry and transitivity of $\sim$ this concludes the proof of equations $(6)_{\text{st,obs}}$ when $k = i$. The proof when $k = j$ is symmetric.

The diagrams in Figures 11, together with the rules (≡-to-∼) and (∼-trans) , provide a slightly different proof of the weak equations $(6)_{\text{st,obs}}$. In these diagrams we use the derived rule (≡-final) which has been proved in example 3.2, and (under the same name) its consequence $\pi_1 \equiv \langle\,\rangle_{\mathbb{1}\times X} : \mathbb{1} \times X \to \mathbb{1}$.

Equation (3) is expressed in the decorated logic as:

$$(3)_{\text{st}}\ \forall i \in Loc,\ u_i^{(2)} \circ \pi_2^{(0)} \circ (u_i \rtimes id_{V_i})^{(2)} \equiv u_i^{(2)} \circ \pi_2^{(0)}$$

which is equivalent to:

$$(3)_{\text{st,obs}}\ \forall k \in Loc, \forall i \in Loc,\ l_k^{(1)} \circ u_i^{(2)} \circ \pi_2^{(0)} \circ (u_i \rtimes id_{V_i})^{(2)} \sim l_k^{(1)} \circ u_i^{(2)} \circ \pi_2^{(0)}$$

We can again split the proof in two cases, with proof trees similar to those for equations $(6)_{\text{st,obs}}$:

1. When $k \neq i$, both sides reduce (in the $\sim$ sense) to $l_k \circ \langle\,\rangle_{V_i \times V_j}$.
2. When $k = i$, both sides reduce (in the $\sim$ sense) to π_2.

Proof Pr_1 :

$$(\sim\text{-subs})\ \dfrac{\overset{\text{A2}}{l_k \circ u_j \sim l_k \circ \langle\,\rangle_{V_j}}}{l_k \circ u_j \circ \pi_2 \circ (u_i \rtimes id_{V_j}) \sim l_k \circ \langle\,\rangle_{V_j} \circ \pi_2 \circ (u_i \rtimes id_{V_j})}$$

Proof Pr_2 :

$$(\equiv\text{-to-}\sim)\ \dfrac{(\equiv\text{-repl})\ \dfrac{(\equiv\text{-trans})\ \dfrac{(\equiv\text{-subs})\ \dfrac{(\sim\text{-to-}\equiv)\ \dfrac{(\text{0-to-1})\ \dfrac{(\sim\text{-trans})\ \dfrac{(\sim\text{-final})\ \dfrac{\langle\,\rangle_{V_j} \circ \pi_2 : \mathbb{1} \times V_j \to \mathbb{1}}{\langle\,\rangle_{V_j} \circ \pi_2 \sim \langle\,\rangle_{\mathbb{1} \times V_j}} \qquad (\sim\text{-sym})\ \dfrac{(\sim\text{-final})\ \dfrac{\pi_1 : \mathbb{1} \times V_j \to \mathbb{1}}{\pi_1 \sim \langle\,\rangle_{\mathbb{1} \times V_j}}}{\langle\,\rangle_{\mathbb{1} \times V_j} \sim \pi_1}}{\langle\,\rangle^{(0)}_{V_j} \circ \pi_2^{(0)} \sim {\pi_1}^{(0)}}}{\langle\,\rangle^{(1)}_{V_j} \circ \pi_2^{(1)} \sim {\pi_1}^{(1)}}}{\langle\,\rangle_{V_j} \circ \pi_2 \equiv \pi_1}}{\langle\,\rangle_{V_j} \circ \pi_2 \circ (u_i \rtimes id_{V_j}) \equiv \pi_1 \circ (u_i \rtimes id_{V_j})} \qquad \overset{\text{P2}}{\pi_1 \circ (u_i \rtimes id_{V_j}) \equiv u_i \circ \pi_1}}{\langle\,\rangle_{V_j} \circ \pi_2 \circ (u_i \rtimes id_{V_j}) \equiv u_i \circ \pi_1}}{l_k \circ \langle\,\rangle_{V_j} \circ \pi_2 \circ (u_i \rtimes id_{V_j}) \equiv l_k \circ u_i \circ \pi_1}}{l_k \circ \langle\,\rangle_{V_j} \circ \pi_2 \circ (u_i \rtimes id_{V_j}) \sim l_k \circ u_i \circ \pi_1}$$

Proof Pr_3 :

$$(\sim\text{-trans})\ \dfrac{(\equiv\text{-to-}\sim)\ \dfrac{(\equiv\text{-subs})\ \dfrac{(\sim\text{-to-}\equiv)\ \dfrac{(\text{0-to-1})\ \dfrac{(\sim\text{-final})\ \dfrac{\langle\,\rangle_{Vi} \circ \pi_1 : V_i \times V_j \to \mathbb{1}}{{\langle\,\rangle_{Vi}}^{(0)} \circ {\pi_1}^{(0)} \sim \langle\,\rangle^{(0)}_{V_i \times V_j}}}{{\langle\,\rangle_{Vi}}^{(1)} \circ {\pi_1}^{(1)} \sim \langle\,\rangle^{(1)}_{V_i \times V_j}}}{\langle\,\rangle_{Vi} \circ \pi_1 \equiv \langle\,\rangle_{V_i \times V_j}}}{l_k \circ \langle\,\rangle_{V_i} \circ \pi_1 \equiv l_k \circ \langle\,\rangle_{V_i \times V_j}}}{l_k \circ \langle\,\rangle_{V_i} \circ \pi_1 \sim l_k \circ \langle\,\rangle_{V_i \times V_j}} \qquad (\sim\text{-subs})\ \dfrac{\overset{\text{A2}}{l_k \circ u_i \sim l_k \circ \langle\,\rangle_{V_i}}}{l_k \circ u_i \circ \pi_1 \sim l_k \circ \langle\,\rangle_{V_i} \circ \pi_1}}{l_k \circ u_i \circ \pi_1 \sim l_k \circ \langle\,\rangle_{Vi \times V_j}}$$

Proof Pr_4 :

$$(\sim\text{-trans})\ \dfrac{(\sim\text{-trans})\ \dfrac{\mathrm{Pr}_1 \qquad \mathrm{Pr}_2}{l_k \circ u_j \circ \pi_2 \circ (u_i \rtimes id_{V_j}) \sim l_k \circ u_i \circ \pi_1} \qquad \mathrm{Pr}_3}{l_k \circ u_j \circ \pi_2 \circ (u_i \rtimes id_{V_j}) \sim l_k \circ \langle\,\rangle_{Vi \times V_j}}$$

Figure 9: Case $k \neq i, j$ (with $i \neq j$)

Proof Pr_5 :

$$\dfrac{\dfrac{\text{A2}}{l_i \circ u_j \sim l_i \circ \langle\rangle_{V_j}}\ (\sim\text{-subs}) \qquad \dfrac{\dfrac{\dfrac{\langle\rangle_{V_j} \circ \pi_2 : V_i \times V_j \to \mathbb{1}}{\langle\rangle^{(0)}_{V_j} \circ \pi^{(0)}_2 \sim \langle\rangle^{(0)}_{1\times V_j}}\ (\sim\text{-final})}{\langle\rangle_{V_j} \circ \pi_2 \equiv \langle\rangle_{1\times V_j}}\ (\sim\text{-to-}\equiv)}{l_i \circ \langle\rangle_{V_j} \circ \pi_2 \equiv l_i \circ \langle\rangle_{1\times V_j}}\ (\equiv\text{-repl}) \quad \dfrac{}{l_i \circ \langle\rangle_{V_j} \circ \pi_2 \sim l_i \circ \langle\rangle_{1\times V_j}}\ (\equiv\text{-to-}\sim)}{l_i \circ u_j \circ \pi_2 \circ (u_i \rtimes id_{V_j}) \sim l_i \circ \langle\rangle_{V_j} \circ \pi_2 \circ (u_i \rtimes id_{V_j})}\ (\sim\text{-subs}) \qquad \dfrac{}{l_i \circ \langle\rangle_{V_j} \circ \pi_2 \sim l_i \circ \langle\rangle_{1\times V_j}}$$

$$(\sim\text{-trans})\ \dfrac{\text{(above)} \qquad \text{(above)}}{l_i \circ u_j \circ \pi_2 \circ (u_i \rtimes id_{V_j}) \sim l_i \circ \langle\rangle_{1\times V_j} \circ (u_i \rtimes id_{V_j})}$$

Proof Pr_6 :

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_1 : \mathbb{1} \times V_j \to \mathbb{1}}{\langle\rangle^{(0)}_{\mathbb{1}\times V_j} \sim \pi^{(0)}_1}\ (\sim\text{-final})}{\langle\rangle^{(1)}_{\mathbb{1}\times V_j} \sim \pi^{(1)}_1}\ (0\text{-to-}1)}{\langle\rangle_{\mathbb{1}\times V_j} \equiv \pi_1}\ (\sim\text{-to-}\equiv)}{\langle\rangle_{1\times V_j} \circ (u_i \rtimes id_{V_j}) \equiv \pi_1 \circ (u_i \rtimes id_{V_j})}\ (\equiv\text{-subs}) \qquad \dfrac{\text{P2}}{\pi_1 \circ (u_i \rtimes id_{V_j}) \equiv u_i \circ \pi_1}$$

$$(\equiv\text{-trans})\ \dfrac{\text{(above)} \qquad \text{(above)}}{\langle\rangle_{1\times V_j} \circ (u_i \rtimes id_{V_j}) \equiv u_i \circ \pi_1}$$

$$(\equiv\text{-subs})\ \dfrac{\langle\rangle_{1\times V_j} \circ (u_i \rtimes id_{V_j}) \equiv u_i \circ \pi_1}{l_i \circ \langle\rangle_{1\times V_j} \circ (u_i \rtimes id_{V_j}) \equiv l_i \circ u_i \circ \pi_1}$$

$$(\equiv\text{-to-}\sim)\ \dfrac{l_i \circ \langle\rangle_{1\times V_j} \circ (u_i \rtimes id_{V_j}) \equiv l_i \circ u_i \circ \pi_1}{l_i \circ \langle\rangle_{1\times V_j} \circ (u_i \rtimes id_{V_j}) \sim l_i \circ u_i \circ \pi_1}$$

Proof Pr_7 :

$$(\sim\text{-trans})\ \dfrac{\mathrm{Pr}_5 \qquad \mathrm{Pr}_6}{l_i \circ u_j \circ \pi_2 \circ (u_i \rtimes id_{V_j}) \sim l_i \circ u_i \circ \pi_1} \qquad \dfrac{\dfrac{\text{A1}}{l_i \circ u_i \sim id_{V_i}}}{l_i \circ u_i \circ \pi_1 \sim \pi_1}\ (\sim\text{-subs})$$

$$(\sim\text{-trans})\ \dfrac{\text{(above)} \qquad \text{(above)}}{l_i \circ u_j \circ \pi_2 \circ (u_i \rtimes id_{V_j}) \sim \pi_1}$$

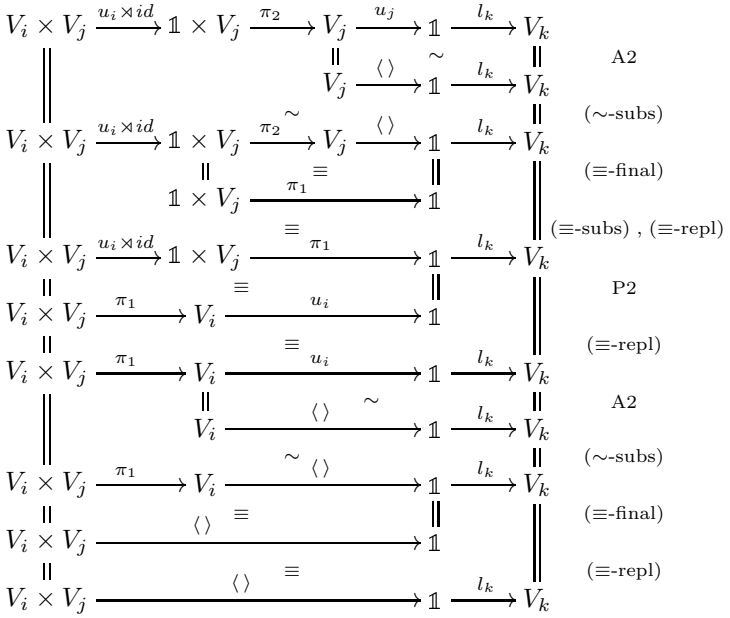
Proof Pr_8 :

$$(\sim\text{-subs})\ \dfrac{\dfrac{\text{A1}}{l_i \circ u_i \sim id_{V_i}}}{l_i \circ u_i \circ \pi_1 \circ (id_{V_i} \ltimes u_j) \sim \pi_1 \circ (id_{V_i} \ltimes u_j)} \qquad \dfrac{\dfrac{\text{P1}}{\pi_1 \circ (id_{V_i} \ltimes u_j) \sim \pi_1}}{\pi_1 \sim \pi_1 \circ (id_{V_i} \ltimes u_j)}\ (\sim\text{-sym})$$

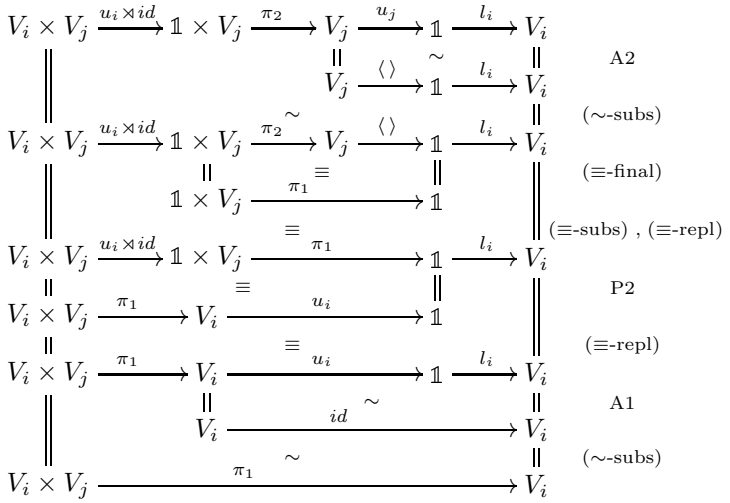
$$(\sim\text{-trans})\ \dfrac{\text{(above)} \qquad \text{(above)}}{l_i \circ u_i \circ \pi_1 \circ (id_{V_i} \ltimes u_j) \sim \pi_1}$$

Figure 10: Case $k = i$ (with $i \neq j$)

Proof Pr_4 :



Proof Pr_7 :



Proof Pr_8 :

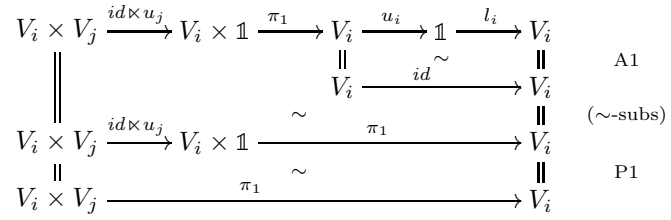


Figure 11: Proofs as diagrams

A.2 Exceptions

Dually, we get the decorated equations for exceptions. In the decorated logic for exceptions, the *coproduct* of two objects A and B is an object $A+B$ with two pure morphisms: the coprojections $\iota_1^{(0)} : A \to A+B$ and $\iota_2^{(0)} : B \to A+B$, which satisfy the usual categorical coproduct property with respect to the pure morphisms. So, as usual, the coprojections $\iota_1^{(0)} : A \to A + \mathbb{0}$ and $\iota_2^{(0)} : B \to \mathbb{0} + B$ are isomorphisms. The *semi-pure coproduct* of a pure morphism $f^{(0)} : A \to C$ and a catcher $g^{(2)} : B \to D$ is a catcher $(f+g)^{(2)} : A+B \to C+D$ (for simplicity we still use the symbol $+$) which is characterized, up to $\equiv$, by the following decorated version of the coproduct property:

$$(f+g)^{(2)} \circ \iota_1^{(0)} \sim \iota_1^{(0)} \circ f^{(0)}$$
$$(f+g)^{(2)} \circ \iota_2^{(0)} \equiv \iota_2^{(0)} \circ g^{(2)}$$

$$\begin{array}{ccc} A & \xrightarrow{f^{(0)}} & C \\ {\scriptstyle \iota_1^{(0)}}\downarrow & \sim & \downarrow{\scriptstyle \iota_1^{(0)}} \\ A+B & \xrightarrow[(f+g)^{(2)}]{} & C+D \\ {\scriptstyle \iota_2^{(0)}}\uparrow & \equiv & \uparrow{\scriptstyle \iota_2^{(0)}} \\ B & \xrightarrow[g^{(2)}]{} & D \end{array}$$

Now, the equations $(6)_{\text{exc}}$ and $(3)_{\text{exc}}$, dual to equations $(6)_{\text{st}}$ and $(3)_{\text{st}}$, can be proved in the dual way.

$$(6)_{\text{exc}} \quad \forall i \neq j \in \mathit{ExCstr},\ (c_i + id_{P_j})^{(2)} \circ \iota_2^{(0)} \circ c_j^{(2)} \equiv (id_{P_i} + c_j)^{(2)} \circ \iota_1^{(0)} \circ c_i^{(2)}$$
$$(3)_{\text{exc}} \quad \forall i \in \mathit{ExCstr},\ (c_i + id_{P_i})^{(2)} \circ \iota_2^{(0)} \circ c_i^{(2)} \equiv \iota_2^{(0)} \circ c_i^{(2)}$$

Proposition 4.10 can be proved by encapsulating these equations, if it is assumed that the coproducts in the decorated logic are coproducts for the propagators. This means that for each propagators $g^{(1)} : A \to C$ and $h^{(1)} : B \to C$ there is a propagator $[g \mid h]^{(1)} : A + B \to C$ which is characterized, up to $\equiv$, by:

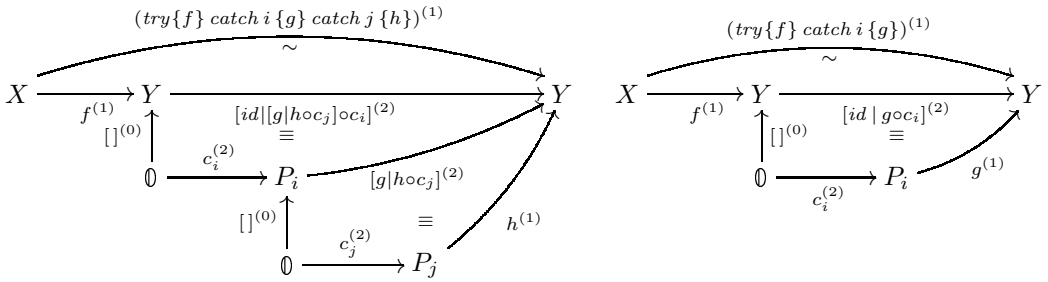
$$[g \mid h]^{(1)} \circ \iota_1^{(0)} \equiv g^{(1)}$$
$$[g \mid h]^{(1)} \circ \iota_2^{(0)} \equiv h^{(1)}$$

Proposition 4.10 states that:

$$(6)_{\text{exc,encaps}} \quad \forall i \neq j \in \mathit{ExCstr}, \mathit{try}\{f\}\ \mathit{catch}\, i\, \{g\}\ \mathit{catch}\, j\, \{h\} \equiv \mathit{try}\{f\}\ \mathit{catch}\, j\, \{h\}\ \mathit{catch}\, i\, \{g\}$$
$$(3)_{\text{exc,encaps}} \quad \forall i \in \mathit{ExCstr},\ \mathit{try}\{f\}\ \mathit{catch}\, i\, \{g\}\ \mathit{catch}\, i\, \{h\} \equiv \mathit{try}\{f\}\ \mathit{catch}\, i\, \{g\}$$

where, according to definition 4.6:

$$\forall i, j \in \mathit{ExCstr},\ \mathit{try}\{f\}\ \mathit{catch}\, i\, \{g\}\ \mathit{catch}\, j\, \{h\} = \triangledown([id \mid [g \mid h \circ c_j] \circ c_i] \circ f)$$
$$\forall i \in \mathit{ExCstr},\ \mathit{try}\{f\}\ \mathit{catch}\, i\, \{g\} = \triangledown([id \mid g \circ c_i] \circ f)$$



Proof of proposition 4.10. It is easy to check that

$$[g \mid h \circ c_j] \equiv [g \mid h] \circ (id_{P_i} + c_j)$$

then it follows from $(6)_{\text{exc}}$ and $(3)_{\text{exc}}$ that

$$\forall i \neq j,\ [g \mid h \circ c_j] \circ c_i \equiv [h \mid g \circ c_i] \circ c_j$$
$$\forall i,\ [g \mid h \circ c_i] \circ c_i \equiv g \circ c_i$$

which implies $(6)_{\text{exc,encaps}}$ and $(3)_{\text{exc,encaps}}$, as required. □