A duality between exceptions and states

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

A duality between exceptions and states

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

October 24., 2011

Abstract

Abstract. In this short note we study the semantics of two basic computational effects, exceptions and states, from a new point of view. In the handling of exceptions we dissociate the control from the elementary operation which recovers from the exception. In this way it becomes apparent that there is a duality, in the categorical sense, between exceptions and states.

Introduction

In this short note we study the semantics of two basic computational effects, exceptions and states, from a new point of view. Exceptions are studied in Section 1. The focus is placed on the exception "flags" which are set when an exception is raised and which are cleared when an exception is handled. We define the *exception constructor* operation which sets the exception flag, and the *exception recovery* operation which clears this flag. States are considered in the short Section 2. Then in Section 3 we show that our point of view yields a surprising result: there exists a symmetry between the computational effects of exceptions and states, based on the categorical duality between sums and products. More precisely, the lookup and update operations for states are respectively dual to the constructor and recovery operations for exceptions. This duality is deeply hidden, since the constructor and recovery operations for exceptions are mixed with the control. This may explain that our result is, as far as we know, completely new.

States and exceptions are *computational effects*: in an imperative language there is no type of states, and in a language with exceptions the type of exceptions which may be raised by a program is not seen as a return type for this program. In this note we focus on the denotational semantics of exceptions and states, so that the sets of states and exceptions are used explicitly. However, with additional logical tools, the duality may be expressed in a way which fits better with the syntax of effects [Dumas et al. 2010].

Other points of view about computational effects, involving monads and Lawvere theories, can be found in [Moggi 1991, Schröder & Mossakowski 2004, Levy 2006, Plotkin & Pretnar 2009]. However it seems difficult to derive from these approaches the duality described in this note.

1 Exceptions

The syntax for exceptions heavily depends on the language. For instance in ML-like languages there are several exception *names*, and the keywords for raising and handling exceptions are `raise` and `handle`, while in Java there are several exception *types*, and the keywords for raising and handling exceptions are `throw` and `try-catch`. In spite of the differences in the syntax, the semantics of exceptions share many similarities. 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,

[*]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`

interchange these two kinds of values: an ordinary value may be "tagged" for constructing an exception, then the "tag" may be cleared in order to recover the value.

First let us focus on the raising of exceptions. Let *Exc* denote the set of *exceptions*. The "tagging" process can be modelled by injective functions $t_i : Par_i \to Exc$ called the *exception constructors*, with disjoint images: for each index i in some set of indices I, the exception constructor $t_i : Par_i \to Exc$ maps a non-exceptional value (or *parameter*) $a \in Par_i$ to an exception $t_i(a) \in Exc$. When a function $f : X \to Y + Exc$ *raises* (or *throws*) an exception of index i, the following *raising* operation is called:

$$raise_{i,Y} : Par_i \to Y + Exc$$

The raising operation $raise_{i,Y}$ is defined as the exception constructor t_i followed by the inclusion of *Exc* in $Y + Exc$.

Given a function $f : X \to Y + Exc$ and an element $x \in X$, if $f(x) = raise_{i,Y}(a)$ for some $a \in Par_i$ then one says that $f(x)$ *raises an exception of index 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 in a unique way as a function which propagates exceptions.

Now let us study the handling of exceptions. The process of clearing the "exception tags" can be modelled by functions $c_i : Exc \to Par_i + Exc$ called the *exception recovery* operations: for each $i \in I$ and $e \in Exc$ the exception recovery operation $c_i(e)$ tests whether the given exception e is in the image of t_i. If this is actually the case, then it returns the parameter $a \in Par_i$ such that $e = t_i(a)$, otherwise it propagates the exception e.

For handling exceptions of indices $i_1, \ldots, i_n$ raised by some function $f : X \to Y + Exc$, one provides a function $g_{i_k} : Par_{i_k} \to Y + Exc$, which may itself raise exceptions, for each k in $\{1, \ldots, n\}$. Then the handling process builds a function which propagates exceptions, it may be named $try\{f\}\ catch\, i_k\{g_k\}_{1 \le k \le n}$ or f *handle* $(i_k \Rightarrow g_k)_{1 \le k \le n}$:

$$f \text{ handle } (i_k \Rightarrow g_k)_{1 \le k \le n} : X + Exc \to Y + Exc$$

Using the recovery operations c_{i_k}, the handling process can be defined as follows.

For each $x \in X + Exc$, $(f \text{ handle } (i_k \Rightarrow g_k)_{1 \le k \le n})(x) \in Y + Exc$ is defined by:
// if x was an exception before the try, then it is just propagated
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
  compute $y := c_{i_k}(y) \in Par_{i_k} + Exc$;
  if $y \in Par_{i_k}$ then return $g_k(y) \in Y + Exc$;
// now y is an exception but it does not have index i_k, for any $k \in \{1, \ldots, n\}$
return $y \in Exc \subseteq Y + Exc$.

Given an exception e of the form $t_i(a)$, the recovery operation c_i returns the non-exceptional value a while the other recovery operations propagate the exception e. This is expressed by the equations (1) in Figure 1. Whenever $Exc = \sum_{i \in I} Par_i$ with the t_i's as coprojections, then equations (1) provide a characterization of the operations c_i's.

2 States

Now let us forget temporarily about the exceptions in order to focus on the semantics of an imperative language. Let *St* denote the set of *states* and *Loc* 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*

For each index $i \in I$:

- a set Par_i (parameters)
- two operations $t_i : Par_i \to Exc$ (exception constructor)
  and $c_i : Exc \to Par_i + Exc$ (exception recovery)
- and two equations:

$$\begin{cases} \forall\, a \in Par_i\,,\ c_i(t_i(a)) = a \in Par_i \subseteq Par_i + Exc \\ \forall\, b \in Par_j\,,\ c_i(t_j(b)) = t_j(b) \in Exc \subseteq Par_i + Exc \quad \text{for every } j \neq i \in I \end{cases} \tag{1}$$

which correspond to commutative diagrams, where m_i and n_i are the injections:

$$\begin{array}{ccc} Par_i + Exc & \xleftarrow{m_i} & Par_i \\ {\scriptstyle c_i}\uparrow & = & \uparrow{\scriptstyle id} \\ Exc & \xleftarrow[t_i]{} & Par_i \end{array} \qquad\qquad \begin{array}{ccccc} Par_i + Exc & \xleftarrow{n_i} & Exc & \xleftarrow{t_j} & Par_j \\ {\scriptstyle c_i}\uparrow & = & & & \uparrow{\scriptstyle id} \\ Exc & & \xleftarrow[t_j]{} & & Par_j \end{array}$$

Figure 1: Semantics of exceptions: constructor and recovery

operation $l_i : St \to Val_i$ for reading the value of location i in the given state. In addition, for each $i \in Loc$ there is an *update* operation $u_i : Val_i \times St \to St$ 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 in Figure 2. Whenever $St = \prod_{i \in Loc} Val_i$ with the l_i's as projections, two states s and s' are equal if and only if $l_i(s) = l_i(s')$ for each i, and equations (2) provide a characterization of the operations u_i's.

For each location $i \in Loc$:

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

$$\begin{cases} \forall\, a \in Val_i\,,\ \forall\, s \in St\,,\ l_i(u_i(a,s)) = a \\ \forall\, a \in Val_i\,,\ \forall\, s \in St\,,\ l_j(u_i(a,s)) = l_j(s) \quad \text{for every } j \neq i \in Loc \end{cases} \tag{2}$$

which correspond to commutative diagrams, where p_i and q_i are the projections:

$$\begin{array}{ccc} Val_i \times St & \xrightarrow{p_i} & Val_i \\ {\scriptstyle u_i}\downarrow & = & \downarrow{\scriptstyle id} \\ St & \xrightarrow[l_i]{} & Val_i \end{array} \qquad\qquad \begin{array}{ccccc} Val_i \times St & \xrightarrow{q_i} & St & \xrightarrow{l_j} & Val_j \\ {\scriptstyle u_i}\downarrow & = & & & \downarrow{\scriptstyle id} \\ St & & \xrightarrow[l_j]{} & & Val_j \end{array}$$

Figure 2: Semantics of states: lookup and update

3 Duality

Our main result is now clear from Figures 1 and 2.

Theorem 3.1. *The duality between categorical products and sums can be extended as a duality between the semantics of the lookup and update operations for states on one side and the semantics of the constructor and recovery operations for exceptions on the other side.*

In [Plotkin & Power 2002] an equational presentation of states is given, with seven families of equations. These equations can be translated in our framework, and it can be *proved* that they are equivalent to equations (2) [Dumas et al. 2010]. Then by duality we get for free seven families of equations for exceptions. For instance, it can be proved that for looking up the value of a location i only the *previous* updating of this location i is necessary, and dually, when throwing an exception constructed with t_i only the *next* recovery operation c_i, with the same index i, is necessary.

References

[Dumas et al. 2010] Jean-Guillaume Dumas, Dominique Duval, Laurent Fousse, Jean-Claude Reynaud. States and exceptions considered as dual effects. arXiv:1001.1662 v4 (2010).

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

[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 & 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).