Math. Struct. in Comp. Science (2012), *vol.* 22, *pp.* 719–722. © Cambridge University Press 2012
doi:10.1017/S0960129511000752 First published online 8 May 2012

A duality between exceptions and states

JEAN-GUILLAUME DUMAS[†], DOMINIQUE DUVAL[†],
LAURENT FOUSSE[†] and JEAN-CLAUDE REYNAUD[‡]

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

Received 24 October 2011

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 that recovers from the exception. In this way it becomes apparent that there is a duality, in the categorical sense, between exceptions and states.

1. Introduction

In this short note we study the semantics of two basic computational effects, exceptions and states, from a new point of view. We discuss exceptions in Section 2. The focus is placed on the exception 'flags', which are set when an exception is raised and then cleared when the exception is handled. We define the *exception constructor* operation that sets the exception flag, and the *exception recovery* operation that clears it. We then consider states in the short Section 3. Finally, in Section 4, 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 look-up and update operations for states are dual to the constructor and recovery operations for exceptions, respectively. 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 that 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 the sets of states and exceptions are used explicitly. However, with additional logical tools, the duality may be expressed in a way that 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 and Mossakowski (2004), Levy (2006) and Plotkin and Pretnar (2009). However, it seems difficult to derive from these approaches the duality described in this note.

2. Exceptions

The syntax for exceptions is heavily dependent on the language. For instance, in ML-like languages there are several exception *names*, and the keywords for raising and handling

https://doi.org/10.1017/S0960129511000752 Published online by Cambridge University Press

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 (that is, non-exceptional) values and the exceptions. It follows that the operations may be classified according to the way they may, or may not, interchange these two kinds of values: an ordinary value may be 'tagged' for constructing an exception, then the 'tag' may be cleared in order to recover the value.

We will begin by focusing 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 $a \in Par_i$ we say that $f(x)$ *raises an exception of index i with parameter a into* Y. We say 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 that propagates exceptions.

Turning now to 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 indeed the case, it returns the parameter $a \in Par_i$ such that $e = t_i(a)$, otherwise it propagates the exception e.

In order to handle the exceptions of indices $i_1, \ldots, i_n$ raised by some function $f : X \to Y + Exc$, we provide 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 that propagates exceptions, called, for example, $try\{f\}\ catch\ i_k\{g_k\}_{1 \leqslant k \leqslant n}$ or $f\ handle\ (i_k \Rightarrow g_k)_{1 \leqslant k \leqslant n}$:

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

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

For each $x \in X + Exc$, $(f\ handle\ (i_k \Rightarrow g_k)_{1 \leqslant k \leqslant n})(x) \in Y + Exc$ 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 $y := c_{i_k}(y) \in Par_{i_k} + Exc$;

https://doi.org/10.1017/S0960129511000752 Published online by Cambridge University Press

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:

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

Fig. 1. Semantics of exceptions: constructor and recovery.

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 Equations (1) in Figure 1. Whenever $Exc = \sum_{i \in I} Par_i$ with the t_i's as coprojections, Equations (1) provide a characterisation of the operations c_i's.

3. States

We will now forget about the exceptions for the moment 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 *look-up* 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 summarised 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 characterisation of the operations u_i's.

4. Duality

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

Theorem 4.1. The duality between categorical products and sums can be extended to a duality between the semantics of the look-up and update operations for states on the one

https://doi.org/10.1017/S0960129511000752 Published online by Cambridge University Press

For each location $i \in Loc$:

— a set Val_i (values)

— two operations $l_i : St \to Val_i$ (look-up)
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:

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

Fig. 2. Semantics of states: look-up and update

hand and the semantics of the constructor and recovery operations for exceptions on the other.

Plotkin and Power (2002) gave an equational presentation of states with seven families of equations. These equations can be translated to our framework, and it can be *proved* that they are equivalent to Equations (2) (Dumas *et al.* 2010). Then, by duality, we get at no cost 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 required, and, dually, when throwing an exception constructed with t_i, only the *next* recovery operation c_i, with the same index i, is required.

References

Dumas, J.-G., Duval, D., Fousse, L. and Reynaud, J.-C. (2010) States and exceptions considered as dual effects. Available at arXiv:1001.1662 v4.

Levy, P. B. (2006) Monads and adjunctions for global exceptions. MFPS 2006. *Electronic Notes in Theoretical Computer Science* **158** 261–287.

Moggi, E. (1991) Notions of Computation and Monads. *Information and Computation* **93** (1) 55–92.

Plotkin, G. D. and Power, J. (2002) Notions of Computation Determine Monads. FoSSaCS 2002. *Springer-Verlag Lecture Notes in Computer Science* **2303** 342–356.

Plotkin, G. D. and Pretnar, M. (2009) Handlers of Algebraic Effects. ESOP 2009. *Springer-Verlag Lecture Notes in Computer Science* **5502** 80–94.

Schröder, L. and Mossakowski, T. (2004) Generic Exception Handling and the Java Monad. AMAST 2004. *Springer-Verlag Lecture Notes in Computer Science* **3116** 443–459.

https://doi.org/10.1017/S0960129511000752 Published online by Cambridge University Press