Scalability using effects

Dominique Duval *

10 June 2013 — SLS 2013 — Extended abstract

Abstract

This note is about using computational effects for scalability. With this method, the specification gets more and more complex while its semantics gets more and more correct. We show, from two fundamental examples, that it is possible to design a deduction system for a specification involving an effect without expliciting this effect.

1 Introduction

A well-known pedagogical trick for teaching complex features is to "lure" the students by first providing a simplified version of this feature, before adding the required corrections to this approximate version. Typically: "*The plural form of most nouns is created by adding the letter 's' to the end of the word, but there are some exceptions...*". In computer science, such an approach is used in the mechanism of *exceptions*, as well as in many other *computational effects*. The aim of this note is to present computational effects as a tool for solving some scalability issues in the design of specifications.

There are several approaches for managing a large specification (or program). For instance, *components* and *modules* can be used for breaking down the specification into smaller pieces; the intended semantics of the whole specification is obtained by "merging" the semantics of its components. Then each module is "right", in the sense that its semantics describes some "part" of the intended semantics. Another method is the *stepwise refinement*, where one builds progressively the required specification from more abstract specifications; during the refinement process, the specification gets more and more complex while its semantics gets more and more precise, until the intended semantics is obtained. Then each step is "right", in the sense that the intended semantics is one of the possible semantics of each intermediate specification.

In this note we focus on using *computational effects* for scalability. With this method, the specification gets more and more complex while its semantics gets more and more *correct*. Thus, each step is "wrong", in the sense that the intended semantics is not a semantics of any intermediate specification.

*Université de Grenoble, Laboratoire Jean Kuntzmann, `Dominique.Duval@imag.fr`.

It follows that the main issue for using computational effects for scalability is whether it is possible to use the intermediate specifications for testing or verifying or proving non-trivial properties of the final specification. In the next sections we argue in favour of a positive answer to this question, from two examples which both rely on a common general algebraic framework. The key point is that each intermediate specification may become "right" simply by classifying its features according to the way they behave with respect to the corresponding effect, without describing explicitly this behaviour.

This note is based on work with J.-C. Reynaud, J.-G. Dumas, L. Fousse and C. Domínguez.

2 Exceptions

The mechanism of *exceptions* has two parts: *raising* exceptions (with keywords `throw` or `raise`) and *handling* them (with keywords `try/catch` or `handle`).

The lure in the raising of exceptions lies in the fact that a function f which takes an argument of type A and returns a value of type B is not interpreted as a map from $[[A]]$ to $[[B]]$ (where $[[T]]$ denotes the interpretation of a type T) but as a map $[[f]] : [[A]] \to [[B]] + \mathit{Exc}$, where Exc is the set of exceptions and "+" denotes the disjoint union. This can be expressed in a categorical framework [9]: a function $f : A \to B$ is interpreted as a map $[[f]] : [[A]] \to [[B]]$ in the Keisli category of the monad $X \mapsto X + \mathit{Exc}$ on the category of sets. The lure in the handling of exceptions is that a function $f : A \to B$ inside a `catch` clause may recover from an exception (it may also raise an exception), so that is interpreted as a map $[[f]] : [[A]] + \mathit{Exc} \to [[B]] + \mathit{Exc}$.

Thus, in order to deal with exceptions, our proposal is to add *decorations* to the syntax: a *catcher* $f^{(2)} : A \to B$ is interpreted as $[[f]] : [[A]] + \mathit{Exc} \to [[B]] + \mathit{Exc}$ while a *propagator* $f^{(1)} : A \to B$ is interpreted as $[[f]] : [[A]] \to [[B]] + \mathit{Exc}$ and a *pure* function $f^{(0)} : A \to B$ is interpreted simply as $[[f]] : [[A]] \to [[B]]$. In order to prove properties of programs involving exceptions, we must also add decorations to the equations: when $f, g : A \to B$ are catchers (which is the general case), a *strong* equation $f \equiv g$ means that $[[f]] = [[g]] : [[A]] + \mathit{Exc} \to [[B]] + \mathit{Exc}$ and a *weak* equation $f \sim g$ means that $[[f]] \circ \mathit{in}_1 = [[g]] \circ \mathit{in}_1 : [[A]] \to [[B]] + \mathit{Exc}$, where $\mathit{in}_1 : [[A]] \to [[A]] + \mathit{Exc}$ is the left coprojection.

A major point is the existence of a *deduction system* associated to these decorations, which can be used for proving properties of specifications using exceptions. For instance, there are rules for the obvious hierarchies: each pure function can be seen as a propagator, each propagator as a catcher, and each strong equation as a weak one. There are also rules expressing the facts that composition preserves the decorations, that the strong equations generate a congruence, and that the weak equations generate a kind of "weak" congruence: if $f_1 \sim f_2$ then $f_1 \circ g \sim f_2 \circ g$ (but in general $h \circ f_1 \not\sim h \circ f_2$). More details can be found in [5].

3 States

Imperative programming relies on the notion of side-effect for states: a state of the memory can be *observed* thanks to `lookup` functions and it can be *modified* thanks to `update` functions.

The lure in this situation is that a function f which takes an argument of type A and returns a value of type B may be interpreted either as $[[f]] : [[A]] \times St \to [[B]]$ if f is an observer or as $[[f]] : [[A]] \times St \to [[B]] \times St$ if f is a modifier (where St is the set of states and "$\times$" the cartesian product). In order to deal with imperative programs, we add *decorations* to the syntax: a *modifier* $f^{(2)} : A \to B$ is interpreted as $[[f]] : [[A]] \times St \to [[B]] \times St$ while an *observer* $f^{(1)} : A \to B$ is interpreted as $[[f]] : [[A]] \times St \to [[B]]$. As with exceptions, a *pure* function $f^{(0)} : A \to B$ is interpreted simply as $[[f]] : [[A]] \to [[B]]$. We also add decorations to the equations: when $f, g : A \to B$ are modifiers (this is the general case), a *strong* equation $f \equiv g$ means that $[[f]] = [[g]] : [[A]] \times St \to [[B]] \times St$ and a *weak* equation $f \sim g$ means that $pr_1 \circ [[f]] = pr_1 \circ [[g]] : [[A]] \times St \to [[B]]$ where $pr_1 : [[B]] \times St \to [[B]]$ is the left projection. This can also be expressed in a categorical framework: an observer $f^{(1)} : A \to B$ is interpreted as a map $[[f]] : [[A]] \to [[B]]$ in the co-Keisli category of the comonad $X \mapsto X \times St$ on the category of sets, while a modifier $f^{(2)} : A \to B$ may be interpreted as a map $[[f]] : [[A]] \to [[B]]$ in the Keisli category of the monad $X \mapsto (X \times St)^{St}$ [9]. For instance, when dealing with a toy class for bank accounts in an object oriented language, the expressions "`deposit(7); balance();`" and "`7 + balance();`" have different effects but they return the same integer. They can be seen as decorated terms $f^{(2)} = \mathtt{balance}^{(1)} \circ \mathtt{deposit}^{(2)} \circ \mathtt{7}^{(0)}$ and $g^{(1)} = \mathtt{+}^{(0)} \circ (\mathtt{7}^{(0)}, \mathtt{balance}^{(1)})$, and indeed it can be proved that $f^{(2)} \sim g^{(1)}$ without introducing any type of states.

In fact, our approach reveals a duality between exceptions and states [3]. Thus, we get rules for proving properties of imperative programs which are dual to the rules mentioned above for exceptions. More details are given in [4]. In this duality, the exception monad $X + E$ corresponds to the comonad $X \times St$, but there is no comonad on sets corresponding to the states monad $(X \times St)^{St}$; from our point of view this is not an issue, in contrast with the point of view of monads and Lawvere theories where there is a need for specific tools for *handlers* [10, 11].

4 Conclusion

In the previous sections we have outlined the construction of deduction systems for dealing with exceptions and states in an *implicitly* way, i.e., without using any "type of exceptions" or "type of states"; we have simply added some decorations to the syntax of the language. This can be done for other computational effects: several examples are given in [6], where we propose a formalization of the fact that the order of evaluation of the arguments of a binary function becomes crucial in presence of effects.

This work relies on categorical tools [2]: mainly on *adjunction*, more precisely on *categories of fractions* [8], and on *limit sketches* [7, 1].

We have described one step in our approach to scalability. Thanks to the categorical framework it should be easy to express the composition of effects, thus getting a stepwise scalability method. The combination of this method with other scalability methods still has to be studied.

References

[1] Michael Barr, Charles Wells. Category Theory for Computing Science, 3rd edition. Publications du Centre de recherches mathématiques, Université de Montréal (1999).

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

[3] Jean-Guillaume Dumas, Dominique Duval, Laurent Fousse, Jean-Claude Reynaud. A duality between exceptions and states. Mathematical Structures in Computer Science 22, p. 719-722 (2012).

[4] Jean-Guillaume Dumas, Dominique Duval, Laurent Fousse, Jean-Claude Reynaud. Decorated proofs for computational effects: States. ACCAT 2012. Electronic Proceedings in Theoretical Computer Science 93, p. 45-59 (2012).

[5] Jean-Guillaume Dumas, Dominique Duval, Laurent Fousse, Jean-Claude Reynaud. Adjunctions for exceptions. arXiv:1207.1255 (2012).

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

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

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

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

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

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