Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 319 (2015) 19–35

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# An Introduction to Algebraic Effects and Handlers Invited tutorial paper

Matija Pretnar[1]

*Faculty of Mathematics and Physics*
*University of Ljubljana*
*Slovenia*

**Abstract**

This paper is a tutorial on algebraic effects and handlers. In it, we explain what algebraic effects are, give ample examples to explain how handlers work, define an operational semantics and a type & effect system, show how one can reason about effects, and give pointers for further reading.

*Keywords:* algebraic effects, handlers, effect system, semantics, logic, tutorial

*Algebraic effects* are an approach to computational effects based on a premise that impure behaviour arises from a set of *operations* such as get & set for mutable store, read & print for interactive input & output, or raise for exceptions [16,18]. This naturally gives rise to *handlers* not only of exceptions, but of any other effect, yielding a novel concept that, amongst others, can capture stream redirection, backtracking, co-operative multi-threading, and delimited continuations [21,22,5].

I keep hearing from people that they are interested in algebraic effects and handlers, but do not know where to start. This is what this tutorial hopes to fix. We will look at how to program with algebraic effects and handlers, how to model them, and how to reason about them. The tutorial requires no special background knowledge except for a basic familiarity with the theory of programming languages (a good introduction can be found in [8,15]).

http://dx.doi.org/10.1016/j.entcs.2015.12.003

| | | | |
|---|---|---|---|
| value $v$ | $::=$ | $x$ | variable |
| | $\mid$ | $\mathbf{true} \mid \mathbf{false}$ | boolean constants |
| | $\mid$ | $\mathbf{fun}\ x \mapsto c$ | function |
| | $\mid$ | $h$ | handler |
| handler $h$ | $::=$ | $\mathbf{handler}\ \{\mathbf{return}\ x \mapsto c_r,$ | (optional) return clause |
| | | $\mathsf{op}_1(x;k) \mapsto c_1, \ldots, \mathsf{op}_n(x;k) \mapsto c_n\}$ | operation clauses |
| computation $c$ | $::=$ | $\mathbf{return}\ v$ | return |
| | $\mid$ | $\mathsf{op}(v; y.\, c)$ | operation call |
| | $\mid$ | $\mathbf{do}\ x \leftarrow c_1\ \mathbf{in}\ c_2$ | sequencing |
| | $\mid$ | $\mathbf{if}\ v\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$ | conditional |
| | $\mid$ | $v_1\, v_2$ | application |
| | $\mid$ | $\mathbf{with}\ v\ \mathbf{handle}\ c$ | handling |

Fig. 1. Syntax of terms.

## 1 Language

Before we dive into examples of handlers, we need to fix a language in which to work. As the order of evaluation is important when dealing with effects, we split language terms (Figure 1) into inert *values* and potentially effectful *computations*, following an approach called *fine-grain call-by-value* [13]. There are a few things worth mentioning:

**Sequencing** In $\mathbf{do}\ x \leftarrow c_1\ \mathbf{in}\ c_2$, we first evaluate $c_1$, and once this returns a value, we bind it to $x$ and proceed by $c_2$. If $x$ does not appear in $c_2$, we abbreviate the sequencing to $c_1; c_2$.

**Operation calls** The call $\mathsf{op}(v; y.\, c)$ passes a *parameter* value $v$ (e.g. the memory location to be read) to the operation $\mathsf{op}$, and after $\mathsf{op}$ performs the effect, its *result* value (e.g. the contents of the memory location) is bound to $y$ and the evaluation of $c$, called a *continuation*, resumes. However, note that encompassing handlers may override this behaviour.

**Generic effects** Having an explicit continuation in the call is convenient for the semantics, but less so for a programmer, who just wants to get back the result of an operation. So, instead of a full-blown operation call, we define a function, called a *generic effect* [18], also labelled as $\mathsf{op}$, which takes a parameter and passes it to an operation call with the trivial continuation:

$$\mathsf{op} \overset{\text{def}}{=} \mathbf{fun}\ x \mapsto \mathsf{op}(x; y.\, \mathbf{return}\ y)$$

Though simpler to use, generic effects are just as expressive because we can recover the operation call $\mathsf{op}(v; y.\, c)$ by evaluating $\mathbf{do}\ y \leftarrow \mathsf{op}\, v\ \mathbf{in}\ c$.

**Language extensions** To focus on new constructs, we shall keep our language small, but for examples, we are going to extend its values with integers, primitive

[1] The material is based upon work supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Award No. FA9550-14-1-0096.