Algebraic effects and handlers (OPLSS 2018 lecture notes)

Andrej Bauer
University of Ljubljana

July 2018

These are the notes and materials for the lectures on algebraic effects and handlers at the Oregon programming languages summer school 2018 (OPLSS). The notes were originally written in Markdown and converted to LaTeX semi-automatically, please excuse strange formatting. You can find all the resources at the accomanying GitHub repository.

The lectures were recorded on video that are available at the summer school web site.

General resources & reading material

- Effects bibliography
- Effects Rosetta Stone
- Programming language Eff

1 What is algebraic about algebraic effects and handlers?

The purpose of the first two lectures is to review algebraic theories and related concepts, and connect them with computational effects. We shall start on the mathematical side of things and gradually derive from it a programming language.

Outline

Pretty much everything that will be said in the first two lectures is written up in "What is algebraic about algebraic effects and handlers?", which still a bit rough around the edges, so if you see a typo please let me know.

Contents of the first two lectures:

- signatures, terms, and algebraic theories
- models of an algebraic theory
- free models of an algebraic theory
- generalization to parameterized operations with arbitrary arities
- sequencing and generic operations
- handlers
- comodels and tensoring of comodels and models

1.1 Problems

Each section contains a list of problems, which are ordered roughly in the order of difficulty, either in terms of trickiness, the amount of work, or prerequisites. I recommend that you discuss the problems in groups, and pick whichever problems you find interesting.

Problem 1.1 (The theory of an associative unital operation)**.** Consider the theory T of an associative operation with a unit. It has a constant ϵ and a binary operation $\cdot$ satisfying equations

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$
$$\epsilon \cdot x = x$$
$$x \cdot \epsilon = x$$

Give a useful description of the free model of T generated by a set X. You can either guess an explicit construction of free models and show that it has the required universal property, or you can analyze the free model construction (equivalence classes of well-founded trees) and provide a simple description of it.

Problem 1.2 (The theory of apocalypse)**.** We formulate an algebraic theory Time in it is possible to explicitly record passage of time. The theory has a single unary operation tick and no equations. Each application of tick records the passage of one time step.
Task: Give a useful description of the free model of the theory, generated by a set X.
Task: Let a given fixed natural number n be given. Describe a theory $\mathsf{Apocalypse}$ which extends the theory Time so that a computation crashes (aborts, necessarily terminates) if it performs more than n of ticks. Give a useful description of its free models.

Advice: do *not* concern yourself with any sort of operational semantics which somehow "aborts" after n ticks. Instead, use equations and postulate that certain computations are equal to an aborted one.

Problem 1.3 (The theory of partial maps)**.** The models of the empty theory are precisely sets and functions. Is there a theory whose models form (a category equivalent to) the category of sets and *partial* functions?

Recall that a partial function $f : A \hookrightarrow B$ is an ordinary function $f : S \to B$ defined on a subset $S \subseteq A$. (How do we define composition of partial functions?)

Problem 1.4 (Models in the category of models)**.** In Example 1.27 of the reading material it is calculated that a model of the theory Group in the category $\mathsf{Mod}(\mathsf{Group})$ is an abelian group. We may generalize this idea and ask about models of theory T_1 in the category of models $\mathsf{Mod}(T_2)$ of theory T_2.

The **tensor product** $T_1 \otimes T_2$ of algebraic theories T_1 and T_2 is a theory such that the category of models of T_1 in the category $\mathsf{Mod}(T_2)$ is equivalent to the category of models of $T_1 \otimes T_2$.

Hint: start by outlining what data is needed to have a T_1-model in $\mathsf{Mod}(T_2)$ is, and pay attention to the fact that the operations of T_1 must be interpreted as T_2-homomorphisms. That will tell you what the ingredients of $T_1 \otimes T_2$ should be.

1.1.1 Problem: Morita equivalence

It may happen that two theories T_1 and T_2 have equivalent categories of models, i.e.,

$$\mathsf{Mod}(T_1) \simeq \mathsf{Mod}(T_2)$$

In such a case we say that T_1 and T_2 are **Morita equivalent**.

Let T be an algebraic theory and t a term in context $x_1, \ldots, x_i$. Define a **definitional extension** $T+(\mathsf{op}{:=}t)$ to be the theory T extended with an additional operation op and equation

$$x_1, \ldots, x_i \mid \mathsf{op}(x_1, \ldots, x_i) = t$$

We say that op is a **defined operation**.
Task: Confirm the intuitive feeling that $T + (\mathsf{op}{:=}t)$ by proving that T and $T + (\mathsf{op}{:=}t)$ are Morita equivalent.
Task: Formulate the idea of a definitional extension so that we allow an arbitrary set of defined operations, and show that we still have Morita equivalence.

Problem 1.5 (The theory of a given set)**.** Given any set A, define the **theory** $T(A)$ **of the set** A as follows:

- for every n and every map $f : A^n \to A$, $\mathsf{op}(f)$ is an n-ary operation
- for all $f : A^i \to A$, $g : A^j \to A$ and $h_1, \ldots, h_i : A^j \to A$, if

  $$f \circ (h_1, \ldots, h_i) = g$$

  then we have an equation

  $$x_1, \ldots, x_j \mid \mathsf{op}(f)(\mathsf{op}(h_1)(x_1, \ldots, x_j), \ldots, h_i(x_1, \ldots, x_j)) = g(x_1, \ldots, x_j)$$

Task: Is $T(\{0, 1\})$ Morita equivalent to another, well-known algebraic theory?

Problem 1.6 (A comodel for non-determinism)**.** In Example 4.6 of the reading material it is shown that there is no comodel of non-determinism in the category of sets. Can you suggest a category in which we get a reasonable comodel of non-determinism?

Problem 1.7 (Formalization of algebraic theories)**.** If you prefer avoiding doing Real Math, you can formalize algebraic theories and their comodels in your favorite proof assistant. A possible starting point is this gist, and a good goal is the construction of the free model of a theory generated by a set (or a type).

Because the free model requires quotienting, you should think ahead on how you are going to do that. Some possibilities are:

- use homotopy type theory and make sure that the types involved are h-Sets
- use setoids
- suggest your own solution

It may be wiser to first show as a warm-up exercise that theories without equations have initial models, as that only requires the construction of well-founded trees (which are inductive types).

2 Designing a programming language

Having worked out algebraic theories in previous lectures, let us turn the equational theories into a small programming language.

What we have to do:

1. Change mathematical terminology to one that is familiar to programmers.
2. Reuse existing concepts (generators, operations, trees) to set up the overall structure of the language.
3. Add missing features, such as primitive types and recursion, and generally rearrange things a bit to make everything look nicer.
4. Provide operational semantics.
5. Provide typing rules.

2.1 Reading material

There are many possible ways and choices of designing a programming language around algebraic operations and handlers, but we shall mostly rely on Matija Pretnar's tutorial An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. A more advanced treatment is available in An effect system for algebraic effects and handlers.

2.2 Change of terminology

- The elements of $\mathsf{Free}_\Sigma(V)$ are are **computations** (instead of trees).
- The elements of V are **values** (instead of generators).
- We speak of **value types** (instead of sets of generators).
- We speak of **computation type** (instead of free models).

Henceforth we ignore equations.

2.3 Abstract syntax

We add only one primitive type, namely bool. Other constructs (integers, products, sums) are left as exercises.
Value:

```
v ::= x                  (variable)
    | false              (boolean constants)
    | true
    | h                  (handler)
    | λ x . c            (function)
```

Handler:

```
h ::= handler { return x ↦ c_ret, ... op_i(x, κ) ↦ c_i, ... }
```

Computation:

```
c ::= return v                  (pure computation)
    | if v then c1 else c2      (conditional)
    | v1 v2                     (application)
    | with v handle c           (handling)
    | do x ← c1 in c2           (sequencing)
    | op (v, λ x . c)           (operation call)
    | fix x . c                 (fixed point)
```

We introduce **generic operations** as syntactic abbreviation and let op v stand for $\mathsf{op}(v, \lambda x.\mathsf{return}\ x)$.

2.4 Operational semantics

We provide small-step semantics, but big step semantics can also be given (see reading material). In the rules below h stands for

```
handler { return x ↦ c_ret, ... op_i(x,y) ↦ c_i, ... }
```

We write e1[e2/x] for e1 with e2 substituted for x. The operational rules are:

```
________________________________
(if true then c1 else c2)  ↦  c1

_________________________________
(if false then c1 else c2)  ↦  c2

______________________
(λ x . c) v  ↦  c[v/x]

_____________________________________
with h handle return v  ↦  c_ret[v/x]

_____________________________________________________________
with h handle op_i(v,κ)  ↦  c_i[v/x, (λ x . with h handle κ x)/y]
```

```

______________________________________
do x ← return v in c₂  ↦  c₂[v/x]

_________________________________________________________________
do x ← op(v, κ) in c₂  ↦  op(v, λ y . do x ← κ y in c₂)

__________________________________
fix x . c  ↦  c[(fix x . c)/x]
```

2.5 Effect system

2.5.1 Value and computation types

Value type:

```
A, B := bool | A → C | C ⇒ D
```

Computation type:

```
C, D := A!Δ
```

Dirt:

```
Δ ::= {op₁, .., opⱼ}
```

The idea is that a computation which returns values of type `A` and *may* perform operations `op₁, ..., opⱼ` has the computation type `A!{op₁, ..., opⱼ}`.

2.5.2 Signature

We presume that some way of declaring operations is given, i.e., that we have a signature Σ which lists operations with their parameters and arities:

```
Σ = { .., opᵢ : Aᵢ ⇝ Bᵢ, ... }
```

Note that the the parameter and the arity types A_i and B_i are both value types.

2.5.3 Typing rules

A typing context assigns value types to free variables:

```
Γ ::= x₁:A₁, .., xᵢ:Aᵢ
```

We think of Γ as a map which takes variables to their types.

There are two forms of typing judgement:

1. $\Gamma \vdash$ `v : A` – value `v` has value type `A` in context Γ
2. $\Gamma \vdash$ `c : C` – computation `c` has computation type `C` in context Γ

Rules for value typing:

```
Γ(x) = A
__________
Γ ⊢ x : A

__________________
Γ ⊢ false : bool

__________________
Γ ⊢ true : bool

```

```
Γ, x : A ⊢ c_ret : B!Θ
Γ, x : Pᵢ, κ : Aᵢ → B!Θ ⊢ cᵢ : B!Θ   (for each opᵢ : Pᵢ ⇝ Aᵢ in Δ)
__________________________________________________________________________
Γ ⊢ (handler { return x ↦ c_ret, ... opᵢ(x) κ ↦ cᵢ, ... }) : A!Δ ⇒ B!Θ

   Γ, x:A ⊢ c : C
______________________
Γ ⊢ (λ x . c) : A → C
```

Rules for computation typing:

```
    Γ ⊢ v : A
__________________
Γ ⊢ return v : A!Δ

Γ ⊢ v : bool    Γ ⊢ c₁ : C    Γ ⊢ c₂ : C
__________________________________________
    Γ ⊢ (if v then c₁ else c₂) : C

Γ ⊢ v₁ : A → C    Γ ⊢ v₂ : A
_____________________________
   Γ ⊢ v₁ v₂ : C

Γ ⊢ v : C ⇒ D     Γ ⊢ c : C
____________________________
 Γ ⊢ (with v handle c) : D

Γ ⊢ c₁ : A!Δ    Γ, x:A ⊢ c₂ : B!Δ
__________________________________
  Γ ⊢ (do x ← c₁ in c₂) : B!Δ

Γ ⊢ v : Aᵢ    opᵢ ∈ Δ    opᵢ : Aᵢ ⇝ Bᵢ
_______________________________________
        Γ ⊢ op v : Bᵢ!Δ

 Γ, x:A ⊢ c : A!Δ
______________________
Γ ⊢ (fix x . c) : A!Δ
```

2.6 Safety theorem

If $\vdash$ `c : A!Δ` then:

1. `c = return v` for some $\vdash$ `v : A` *or*
2. `c = op(v, κ)` for some `op` $\in\Delta$ and some value `v` and continuation κ, *or*
3. `c` $\mapsto$`c'` for some $\vdash$ `c' : A!Δ`.

For a mechanised proof see An effect system for algebraic effects and handlers.

2.7 Other considerations

The effect system suffers from the so-called *poisoning*, which can be resolved if we introduce **effect subtyping**. Recursion requires that we use domain-theoretic denotational semantics. Such a semantics turns out to be adequate (but not fully abstract for the same reasons that domain theory is not fully abstract for PCF). See An effect system for algebraic effects and handlers where the above points are treated carefully.

2.7.1 Problems

Problem 2.1 (Products)**.** Add simple products `A ×B` to the core language:

1. Extend the syntax of values with pairs.
2. Extend the syntax of computations with an elimination of pairs, e.g., `do (x,y) ←`c_1` in `c_2.
3. Extend the operational semantics.
4. Extend the typing rules.

Problem 2.2 (Sums)**.** Add simple sums `A + B` to the core language:

1. Extend the syntax of values with injections.
2. Extend the syntax of computations with an elimination of sums (a suitable `match` statement).
3. Extend the operational semantics.
4. Extend the typing rules.

Add the `empty` and `unit` types to the core language. Follow the same steps as in the previous exercises.

Problem 2.4 (Non-terminating program)**.** Define a program which prints infinitely many booleans. You may assume that the `print : bool →unit` operation is handled appropriately by the runtime environment. For extra credit, make it "funny".

Problem 2.5 (Implementation)**.** Implement the core language from Matija Pretnar's tutorial. To make it interesting, augment it with recursive function definitions, integers, and product types. Consider implementing the language as part of the Programming Languages Zoo.

3 Programming with algebraic effects and handlers

In the last lecture we shall explore how algebraic operations and handlers can be used in programming.

3.1 Eff

There are several languages that support algebraic effects and handlers. The ones most faithful to the theory of algebraic effects are Eff and the multicore OCaml. They have very similar syntax, and we could use either, but let us use Eff, just because it was the first language with algebraic effects and handlers.

You can run Eff in your browser or install it locally. The page also has a quick overview of the syntax of Eff, which mimics the syntax of OCaml.

3.2 Reading material

We shall draw on examples from An introduction to algebraic effects and handlers and Programming with algebraic effects and handlers. Some examples can be seen also at the Effects Roset Stone.

Other examples, such as I/O and redirection can be seen at the try Eff page.

3.3 Basic examples

Exceptions

```
effect Abort : unit -> empty

let example b =
  handle
    let x = 7 in
    let y = 8 in
    if b then
      (match perform (Abort ()) with)
    else
      x + y
  with
  | v -> v + 20
  | effect (Abort ()) _ -> 42
```

State

```
(** State *)

effect Get : unit -> int
effect Set : int -> unit

(* The standard state handler. *)
let state' = handler
  | v -> (fun _ -> v)
  | effect (Get ()) k -> (fun s -> (k s) s)
  | effect (Set s') k -> (fun _ -> (k ()) s')
;;

let example1 () =
  let f =
    (with state' handle
      let x = perform (Get ()) in
      perform (Set (2 * x)) ;
      perform (Get ()) + 10)
  in
  f 30
;;

(* Better state handler, using finally clause *)
let state initial = handler
  | y -> (fun _ -> y)
  | effect Get k -> (fun s -> k s s)
  | effect (Set s') k -> (fun _ -> k () s')
  | finally f -> f initial
;;

let example2 () =
  with state 30 handle
    let x = perform (Get ()) in
    perform (Set (2 * x)) ;
    perform Get + 10
```

3.4 Multi-shot handlers

A handler has access to the continuation, and it may do with it whatever it likes. We may distinguish handlers according to how many times the continuation is invoked:

- an **exception-like** handler does not invoke the continuation
- a **single-shot** handler invokes the continuation exactly once

- a **multi-shot** handler invokes the continuation more than once

Of course, combinations of these are possible, and there are handlers where it's difficult to "count" the number of invocations of the continuation, such as multi-threading below.

An exception-like handler is, well, like an exception handler.

A single-shot handler appears to the programmer as a form of dynamic-dispatch callbacks: performing the operation is like calling the callback, where the callback is determined dynamically by the enclosing handlers.

The most interesting (and confusing!) are multi-shot handlers. Let us have a look at one such handler.

3.4.1 Ambivalent choice

Ambivalent choice is a computational effect which works as follows. There is an exception Fail : unit $\to$ empty which signifies failure to compute successfully, and an operation Select : α list $\to$ α, which returns one of the elements of the list. It has to do return an element such that the subsequent computation does *not* fail (if possible).

With ambivalent choice, we may solve the n-queens problem (of placing n queens on an $n \times n$ chess board so they do not attack each other):

```
(* The queens problem using ambivalent choice. *)

type queen = int * int

effect Select : int list -> int
effect Fail : unit -> empty

(* Do the given queens attack each other? *)
let no_attack (x,y) (x',y') =
  x <> x' && y <> y' && abs (x - x') <> abs (y - y')
;;

(* Given that queens qs are already placed, return the list of
   rows in column x which are not attacked yet. *)
let available x qs =
  filter (fun y -> forall (no_attack (x,y)) qs) [1;2;3;4;5;6;7;8]
;;

(* Solve the queens problem by guessing what to do *)
let queens () =
  let rec place x qs =
    if x = 9 then
      qs
    else
      let y = perform (Select (available x qs)) in
      place (x+1) ((x,y) :: qs)
  in
  place 1 []

(* A handler for ambivalent choice which uses depth-first search *)
let dfs = handler
  | v -> v
  | effect (Select lst) k ->
     let rec tryem = function
       | [] -> (match perform (Fail ()) with)
       | x::xs -> (handle k x with effect (Fail ()) _ -> tryem xs)
     in
     tryem lst
;;
```

```

(* A handler for ambivalent choice which uses depth-first search *)
let dfs_all = handler
  | v -> [v]
  | effect (Select lst) k ->
     let rec tryem = function
       | [] -> []
       | x::xs -> (handle k x with
                   | lst -> lst @ (tryem xs)
                   | effect (Fail ()) _ -> tryem xs)
     in
     tryem lst
;;

(* And we can solve the problem: *)
let solution =
  with dfs handle queens ()
;;

let all_solutions =
  with dfs_all handle queens ()
```

3.5 Cooperative multi-threading

Operations and handlers have explicit access to continuations. A handler need not invoke a continue, it may instead store it somewhere and run *another* (previously stored) continuation. This way we get *threads*.

```
(* This example is described in Section 6.10 of "Programming with Algebraic Effects and
   Handlers" by A. Bauer and M. Pretnar. *)

type thread = unit -> unit

effect Yield : unit -> unit
effect Spawn : thread -> unit

(* We will need a queue to keep track of inactive threads.
   We implement the queue as state. *)

effect Dequeue : unit -> thread option
effect Enqueue : thread -> unit

(* The queue handler *)
let queue initial = handler
  | effect (Dequeue ()) k ->
    (fun queue -> match queue with
       | [] -> k None []
       | hd::tl -> k (Some hd) tl)
  | effect (Enqueue y) k -> (fun queue -> k () (queue @ [y]))
  | x -> (fun _ -> x)
  | finally x -> x initial
;;

(* Round-robin thread scheduler. It is an example of a recursively defined handler. *)
let round_robin =
  let dequeue_thread () =
    match perform (Dequeue ()) with
    | None -> ()
    | Some t -> t ()
  in
  let rec h () = handler
    | effect Yield k -> perform (Enqueue k) ; dequeue_thread ()
    | effect (Spawn t) k -> perform (Enqueue k) ; with h () handle t ()
    | () -> dequeue_thread ()
  in
  h ()
;;

(* An example of nested multithreading. We have a thread which prints
   the letter a and another one which has two sub-threads printing x and y. *)

let print_list lst =
  iter (fun x -> perform (Print x) ; perform Yield) lst
;;

with queue [] handle
with round_robin handle
 perform (Spawn (fun _ -> print_list ["a"; "b"; "c"; "d"; "e"])) ;
 perform (Spawn (fun _ -> print_list ["A"; "B"; "C"; "D"; "E"]))
;;


(* We can run an unbounded amount of threads. The following example enumerates all
   reduced positive fractions less than 1 by spawning a thread for each denominator
   between d and e. *)

let rec fractions d e =
  let rec find_fractions n =
    (* If the fraction is reduced, print it and yield *)
    if gcd n d = 1 then
      perform (Print (to_string n ^ "/" ^ to_string d ^ ", ")); perform Yield
    else ();
    if d > n then
      find_fractions (n+1)
    else ()
  in
  (* Spawn a thread for the next denominator *)
  (if d < e then
     perform (Spawn (fun _ -> perform Yield; fractions (d + 1) e)) else ()) ;
  (* List all the fractions with the current denominator *)
  find_fractions 1
;;

with queue [] handle
with round_robin handle
 fractions 1 100
;;
```

3.6 Tree representation of a functional

Suppose we have a **functional**

$$h : (\mathsf{int} \to \mathsf{bool}) \to \mathsf{bool}$$

When we apply it to a function $f : \mathsf{int} \to \mathsf{bool}$, we feel that $h\ f$ will proceed as follows: h will *ask* f about the value $f\ x_0$ for some integer x_0. Depending on the result it gets, it will then ask some furter question $f\ x_1$, and so on, until it provides an *answer* a.

We may therefore represent such a functional h as a **tree**:

- the leaves are the answers
- a node is labeled by a question, which has two subtrees representing the two possible continuations (depending on the answer)

We may encode this as the datatype:

```
type tree =
  | Answer of bool
  | Question of int * tree * tree
```

Given such a tree, we can recreate the functional h:

```
let rec tree2fun t f =
  match t with
  | Answer y -> y
  | Question (x, t1, t2) -> tree2fun (if f x then t1 else t2) f
```

Can we go backwards? Given h, how do we get the tree? It turns out this is not possible in a purely functional setting in general (but is possible for out specific case because int $\to$ bool is *compact*, Google "impossible functionals"), but it is with computational effects.

```
(** This code is compatible with Eff 5.0, see http://www.eff-lang.org *)

(** We show that with algebraic effects and handlers a total functional
    [(int -> bool) -> bool] has a tree representation. *)

(* A tree representation of a functional. *)
type tree =
  | Answer of bool
  | Question of int * tree * tree

(** Convert a tree to a functional. *)
let rec tree2fun t a =
  match t with
  | Answer y -> y
  | Question (x, t1, t2) -> tree2fun (if a x then t1 else t2) a

(** An effect that we will use to report how the functional is using its argument. *)
effect Report : int -> bool

(** Convert a functional to a tree. *)
let rec fun2tree h =
  handle
    Answer (h (fun x -> perform (Report x)))
  with
  | effect (Report x) k -> Question (x, k false, k true)

let example1 = fun2tree (fun f -> true)

let example2 = fun2tree (fun f -> f 10; true)

let example3 = fun2tree (fun f -> if f 10 then (f 30 || f 15) else (f 20 && not (f 8)))

(* This one is pretty large, so take care *)
let example4 =
  (* convert a string of booleans to an int *)
  let rec to_int = function
    | [] -> 0
    | b :: bs -> (if b then 1 else 0) + 2 * to_int bs
  in
  fun2tree (fun a -> a (to_int [a 0; a 1; a 2; a 3; a 4; a 5; a 6; a 7; a 8]))
```

3.7 Problems

Problem 3.1 (Breadth-first search)**.** Implement the *breadth-first search* strategy for ambivalent choice.

Problem 3.2 (Monte Carlo sampling)**.** The online Eff page has an example showing a handler which modifies a probabilistic computation (one that uses randomness) to one that computes the *distribution* of results. The handler computes the distribution in an exhaustive way that quickly leads to inefficiency.

Improve it by implement a Monte Carlo handler for estimating distributions of probabilistic computations.

Problem 3.3 (Recursive cows)**.** Contemplate the recursive cows.