

An Introduction to Algebraic Effects and Handlers

Invited tutorial paper

Matija Pretnar¹

*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]).

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*,

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.

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

Fig. 1. Syntax of terms.

following an approach called *fine-grain call-by-value* [13]. There are a few things worth mentioning:

Sequencing In **do** $x \leftarrow c_1$ **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 **op**($v; y. c$) passes a *parameter* value v (e.g. the memory location to be read) to the operation **op**, and after **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 **op**, which takes a parameter and passes it to an operation call with the trivial continuation:

$$\text{op} \stackrel{\text{def}}{=} \text{fun } x \mapsto \text{op}(x; y. \text{return } y)$$

Though simpler to use, generic effects are just as expressive because we can recover the operation call **op**($v; y. c$) by evaluating **do** $y \leftarrow \text{op } v$ **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 arithmetic functions, strings, recursive functions **rec fun** $f x \mapsto c$, the unit $()$ and pairs (v_1, v_2) . Furthermore, we allow patterns in binding constructs (functions, handler clauses, operation calls, and sequencing). In particular, we use the pattern $_$ to denote ignored parameters, and a pair pattern (x_1, x_2) to extract components from a pair. For example, we bind 7 to x and ignore 8 in the application **(fun** $(x, _)$ $\mapsto 6 + x)$ $(7, 8)$.

Separation of values & computations We were a bit lax about the separation of values and computations when writing the last example. Since the addition $6 + x$ is in fact a double application $((+) 6) x$, the first application $(+) 6$ is already

a computation. Thus, it cannot be applied to x because both subterms of an application must be values. Instead, we need to use sequencing and write the example in our restricted syntax as:

$$(\mathbf{fun} (x, _)\mapsto \mathbf{do} f \leftarrow (+) 6 \mathbf{in} f x) (7, 8)$$

However, this longer form adds little value and makes examples hard to read, so while keeping it in mind, we are going to use the shorter form from now on.

Conversely, we shall implicitly insert **return** whenever we use a value where a computation is expected. For example, we shall write $\mathbf{fun} x \mapsto \mathbf{fun} y \mapsto (x, y)$ instead of $\mathbf{fun} x \mapsto \mathbf{return} (\mathbf{fun} y \mapsto \mathbf{return} (x, y))$.

Semantics Observe that each operation call creates a branching point in the evaluation, with as many branches as there are possible results that can be yielded to the continuation. For example, **decide** will have two branches, **print** just one, and **read** will have infinite many branches: one for each possible input. Thus, we can imagine computations as trees, whose leaves are returned values and branching points are called operations. For an example, see Figure 2.

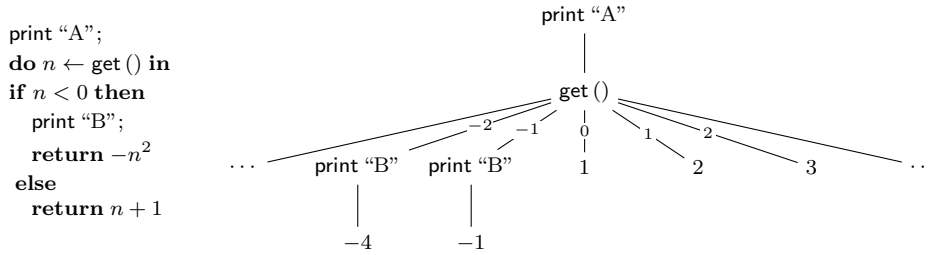


Fig. 2. A computation and a corresponding tree.

In the presence of recursion, some of the leaves of the tree may also be labelled by \perp to indicate a divergent computation that does not call any operations. A divergent computation that repeatedly calls operations is represented by a non-well-founded tree. Denotational semantics is further discussed in Section 6.3.

2 Examples

We now informally describe the behaviour of handlers through examples. You may also prefer to first take a look at the operational semantics given in Section 3.

2.1 Input & output

Let us start with input & output as it is a very simple algebraic effect, but one which exposes almost all important aspects of handlers. It can be described by two operations: **print**, which takes a message to be printed and yields the unit value $()$, and **read**, which takes a unit value and yields a string that was read. For example, a computation that asks the user for his forename and surname and prints out his

full name, is written as:

```
printFullName  $\stackrel{\text{def}}{=} \text{print "What is your forename?";}$ 
              do forename  $\leftarrow \text{read} ()$  in
              print "What is your surname?";
              do surname  $\leftarrow \text{read} ()$  in
              print (join forename surname)
```

where `join` is a function that takes two strings and joins them with a space in the middle.

2.1.1 Constant input

A simple example of a handler is:

```
handler {read( $\_;$ ; k)  $\mapsto k$  "Bob"}
```

which provides a constant input string “Bob” each time `read` is called. We can, of course, generalise it to a function that takes a string s and returns a handler that feeds it to `read`:

```
alwaysRead  $\stackrel{\text{def}}{=} \text{fun } s \mapsto \text{handler } \{\text{read}(\_;$  k)  $\mapsto k s\}$ 
```

This handler works as follows: whenever `read` is called, we ignore its unit parameter and capture its continuation in a function k that expects the resulting string and resumes the evaluation when applied. Next, instead of calling `read`, we evaluate the computation in the handling clause: we resume the continuation k , but instead of reading the string from interactive input, we yield the constant string s . The handler implicitly continues to handle the continuation, so any `read` in the handled computation again yields s . If the handled computation calls any operation other than `read`, the call escapes the handler, but the handler again wraps itself around the continuation so that it may handle any further `read` calls. For example, evaluating

```
with (alwaysRead "Bob") handle printFullName
```

first prints out “What is your name?” as `print` is unhandled. Then, `read` is handled so “Bob” gets bound to *forename*. Similarly, the second `print` is unhandled, and in the second `read`, “Bob” gets bound to *surname* as well and finally “Bob Bob” is printed out.

It is not obvious whether handlers should continue handling operations in the continuation, or handle just the first call. Experience with exception handlers offer us no guidance here, because raised exceptions have no continuation, and so the two choices are equivalent. As it turns out, the first choice, which we are settling on in this paper, has nicer denotational semantics, is what one usually desires in practice, and is perhaps also more intuitive because `with h handle c` suggests that the whole c should be handled by h . The second choice leads to *shallow handlers* [10], which are more convenient for certain uses, and can be considered a more elementary approach as they can express the usual handlers through recursion.

2.1.2 Reversed output

We can use handlers to not only change what is fed to the continuation, but also to change the way the continuation is used. For example, to reverse the order of printouts, we use:

$$\text{reverse} \stackrel{\text{def}}{=} \text{handler } \{\text{print}(s; k) \mapsto k (); \text{print } s\}$$

Here, we handle a `print` by first calling the continuation, and only after this is finished, print out `s`. Since the handler wraps itself around `k`, the same rule applies for the continuation and so all printouts are reversed. So, if we define

$$\text{abc} \stackrel{\text{def}}{=} \text{print "A"; print "B"; print "C"}$$

then `with reverse handle abc` prints out first “C”, then “B”, and finally “A”.

2.1.3 Collecting output

A more useful handler is one that collects all printouts into one big string and returns it together with the final value:

$$\begin{aligned} \text{collect} \stackrel{\text{def}}{=} \text{handler } \{ & \text{return } x \mapsto \text{return } (x, "") \\ & \text{print}(s; k) \mapsto \\ & \quad \text{do } (x, \text{acc}) \leftarrow k () \text{ in} \\ & \quad \text{return } (x, \text{join } s \text{ acc}) \} \end{aligned}$$

If the handled computation does not print anything and just returns some value `x`, we need to handle it by returning an empty string in addition to `x`. But if a computation prints some string `s`, we resume the continuation. Since this is handled in the same way, it returns the accumulated string `acc` in addition to the final value `x`. Now, we only need to join `s` with `acc` and return it together with `x`. If we handle `abc` with `collect`, we get a pair `((), “A B C”)`, where `()` is the unit result of the last `print`.

We can also nest handlers, and

$$\text{with collect handle (with reverse handle abc)}$$

evaluates to `((), “C B A”)`. The order in which we nest the handlers is significant as it is the innermost handler that determines how to first handle the call. If we switch the handlers in the above example, we get `((), “A B C”)` because `collect` handles all `print` calls, and so none reach the `reverse` handler, which then does nothing.

Alternatively, we could implement the same handler using a technique called *parameter-passing* [22], where we transform the handled computation into a function that passes around a parameter, in our case the accumulated string:

$$\begin{aligned} \text{collect}' \stackrel{\text{def}}{=} \text{handler } \{ & \text{return } x \mapsto \text{fun } \text{acc} \mapsto \text{return } (x, \text{acc}) \\ & \text{print}(s; k) \mapsto \\ & \quad \text{fun } \text{acc} \mapsto (k ()) (\text{join } \text{acc } s) \} \end{aligned}$$

When a computation returns a value x , there will be no further printouts, so we can return the given accumulator acc in addition to x . But if `print` is called, we resume the continuation by yielding it the expected unit result. Since the continuation is further handled into a function, we need to pass $k()$ the new accumulator, which is acc extended with s . To obtain the collected output of a computation c , we apply the resulting function to the empty accumulator as:

(**with collect' handle** c) “”

In Section 5, we show that `collect` and `collect'` indeed exhibit equivalent behaviour. Using parameter-passing, we can also implement a converse handler that feeds words from a given string to the input.

2.2 Exceptions

Exception handlers are, of course, a special instance of handlers. We represent exceptions with an operation `raise` that takes an exception argument (e.g. error message) and yields nothing to the continuation (for more details on how this can be enforced, see Example 4.1).

In practice, exception handlers are rarely reused, but an example of a more general exception handler is:

$$\text{default} \stackrel{\text{def}}{=} \mathbf{fun} \ x \mapsto \mathbf{handler} \ \{\text{raise}(_; _) \mapsto \mathbf{return} \ x\}$$

which returns a default value x in case the handled computation raises an exception.

2.3 Non-determinism

Handlers can be used not only to override existing effectful behaviour, but to define new one as well. To implement non-determinism, we take a single operation `decide`, which takes a unit parameter, and non-deterministically yields a boolean. Then, a binary choice can be implemented as a function

$$\begin{aligned} \text{choose} \stackrel{\text{def}}{=} \mathbf{fun} \ (x, y) \mapsto \\ \mathbf{do} \ b \leftarrow \text{decide} \ () \ \mathbf{in} \\ \mathbf{if} \ b \ \mathbf{then} \ (\mathbf{return} \ x) \ \mathbf{else} \ (\mathbf{return} \ y) \end{aligned}$$

However, unlike `print`, we assume no intrinsic behaviour for `decide`, and we must use handlers to determine whether to return a fixed result, a random result, an optimal result, or all results. Without an encompassing handler, an application `choose(3, 4)` is stuck when it encounters the `decide` call. The simplest handler for `decide` is

$$\text{pickTrue} \stackrel{\text{def}}{=} \mathbf{handler} \ \{\text{decide}(_; k) \mapsto k \ \mathbf{true}\}$$

which makes each **decide** yield **true** to the continuation, so **choose** always chooses the left argument. So, if we define

```
chooseDiff  $\stackrel{\text{def}}{=} \mathbf{do} \ x_1 \leftarrow \mathbf{choose} \ (15, 30) \ \mathbf{in}$ 
   $\mathbf{do} \ x_2 \leftarrow \mathbf{choose} \ (5, 10) \ \mathbf{in}$ 
   $\mathbf{return} \ (x_1 - x_2)$ 
```

then **with pickTrue handle chooseDiff** will choose 15 for x_1 and 5 for x_2 , and will thus evaluate to **return 10**.

2.3.1 Maximal result

With handlers, we can also traverse all possible branches to select the maximal result:

```
pickMax  $\stackrel{\text{def}}{=} \mathbf{handler} \ \{\mathbf{decide}(\_ ; k) \mapsto$ 
   $\mathbf{do} \ x_t \leftarrow k \ \mathbf{true} \ \mathbf{in}$ 
   $\mathbf{do} \ x_f \leftarrow k \ \mathbf{false} \ \mathbf{in}$ 
   $\mathbf{return} \ \max(x_t, x_f)\}$ 
```

In this case, evaluating **with pickTrue handle chooseDiff** will make the choices needed to get the maximal possible difference 25, even if this means choosing the smaller argument of **choose** (in particular, we pick 30 for x_1 and 5 for x_2).

If we included lists in our language, we could adapt **pickMax** to a handler **pickAll** that select all possible results [5]. To do so, the return clause would return a singleton list containing the returned value, while the **decide** clause would concatenate the lists x_t and x_f that result from yielding both possible results to the handled continuation.

2.3.2 Backtracking

To implement backtracking, where we employ non-deterministic search for a given solution, we add an operation **fail** to signify that no solution exists. Then, for example:

```
rec fun choesInt (m, n)  $\mapsto$ 
  if m > n then fail () else
  do b  $\leftarrow$  decide () in
  if b then (return m) else choesInt (m + 1, n)
```

is a function that non-deterministically chooses an integer in the interval $[m, n]$, or fails if this interval is empty, while:

```
pythagorean  $\stackrel{\text{def}}{=} \mathbf{fun} \ (m, n) \ \mapsto$ 
  do a  $\leftarrow$  choesInt (m, n - 1) in
  do b  $\leftarrow$  choesInt (a + 1, n) in
  if isSquare (a2 + b2) then (return (a, b,  $\sqrt{a^2 + b^2}$ )) else fail ()
```

is a function that searches for an integer Pythagorean triple (a, b, c) such that $m \leq a < b \leq n$. We perform backtracking by handling each `decide` by first trying to yield `true`, and if this fails, yield `false`:

$$\begin{aligned} \text{backtrack} &\stackrel{\text{def}}{=} \text{handler } \{\text{decide}(_ ; k) \mapsto \\ &\quad \text{with} \\ &\quad \quad \text{handler } \{\text{fail}(_ ; _) \mapsto k \text{ false}\} \\ &\quad \text{handle} \\ &\quad \quad k \text{ true}\} \end{aligned}$$

Then, `with backtrack handle pythagorean` (m, n) finds $(5, 12, 13)$ for $(m, n) = (4, 15)$ but fails for $(m, n) = (7, 10)$. The exact triple found depends on the implementation of the handler. If, instead, we first tried yielding `false`, the resulting triple for $(m, n) = (4, 15)$ would be $(9, 12, 15)$. To get a list of all possible triples, we can use the handler `pickAll` from Section 2.3.1, but extended with a clause that handles `fail` with an empty list.

2.4 State

We represent state with operations `set` for setting the state contents, and `get` for reading them. For simplicity, we assume a single memory location that holds an integer. So, `set` takes an integer, stores it, and returns a unit result, while `get` takes a unit parameter, reads the stored integer, and returns it.

We can use handlers to temporarily alter the stored value or to log all updates. But we can also use them to implement stateful behaviour even if we do not assume a built-in one. Like in Section 2.1.3, we use a parameter-passing handler to pass around the current state:

$$\begin{aligned} \text{state} &\stackrel{\text{def}}{=} \text{handler } \{\text{get}(_ ; k) \mapsto \text{fun } s \mapsto (k \ s) \ s \\ &\quad \text{set}(s ; k) \mapsto \text{fun } _ \mapsto (k \ ()) \ s \\ &\quad \text{return } x \mapsto \text{fun } _ \mapsto \text{return } x\} \end{aligned}$$

We handle `get` with a function that takes the current state s and passes it first as a result of `get` to the continuation, and then again as the unchanged state. Conversely, we handle `set` by first yielding the unit result, and then applying the handled continuation to the new state s as given in the parameter of `get`.

The return clause of `state` ignores the final state, but if we want to inspect it, we can return it together with the final value by changing the return clause to:

$$\text{return } x \mapsto \text{fun } s \mapsto \text{return } (s, x)$$

2.4.1 Transactions

In a similar way, we can implement transactional memory, where we commit the changed state only after the handled computation successfully terminated with a

value, so in case an exception is raised, the memory contents remain unchanged:

$$\text{transaction} \stackrel{\text{def}}{=} \text{handler } \{ \text{get}(_, k) \mapsto \text{fun } s \mapsto (k \ s) \ s \\ \text{set}(s; k) \mapsto \text{fun } _ \mapsto (k \ ()) \ s \\ \text{return } x \mapsto \text{fun } s \mapsto \text{set } s; \text{return } x \}$$

3 Operational semantics

To make the intuition about the behaviour of computations concrete, we now give an operational semantics. The idea behind it is that operation calls do not perform actual effects (e.g. printing to an output device), but behave as signals that propagate outwards until they reach a handler with a matching clause. For simplicity, any operation call that escapes all handlers will be treated as a terminating computation, i.e. one that does not further reduce. We can assume that actual effectful behaviour is simulated by an outermost handler, or consider one of the approaches listed in Section 6.5.

$$\frac{c_1 \rightsquigarrow c'_1}{\text{do } x \leftarrow c_1 \text{ in } c_2 \rightsquigarrow \text{do } x \leftarrow c'_1 \text{ in } c_2} \qquad \frac{}{\text{do } x \leftarrow \text{return } v \text{ in } c \rightsquigarrow c[v/x]}$$

$$\frac{}{\text{do } x \leftarrow \text{op}(v; y. c_1) \text{ in } c_2 \rightsquigarrow \text{op}(v; y. \text{do } x \leftarrow c_1 \text{ in } c_2)} \qquad \frac{}{\text{if true then } c_1 \text{ else } c_2 \rightsquigarrow c_1}$$

$$\frac{}{\text{if false then } c_1 \text{ else } c_2 \rightsquigarrow c_2} \qquad \frac{}{(\text{fun } x \mapsto c) v \rightsquigarrow c[v/x]}$$

In the following rules, we set $h = \text{handler } \{ \text{return } x \mapsto c_r, \text{op}_1(x; k) \mapsto c_1, \dots, \text{op}_n(x; k) \mapsto c_n \}$:

$$\frac{c \rightsquigarrow c'}{\text{with } h \text{ handle } c \rightsquigarrow \text{with } h \text{ handle } c'} \qquad \frac{}{\text{with } h \text{ handle } (\text{return } v) \rightsquigarrow c_r[v/x]}$$

$$\frac{}{\text{with } h \text{ handle } \text{op}_i(v; y. c) \rightsquigarrow c_i[v/x, (\text{fun } y \mapsto \text{with } h \text{ handle } c)/k] \quad (1 \leq i \leq n)}$$

$$\frac{}{\text{with } h \text{ handle } \text{op}(v; y. c) \rightsquigarrow \text{op}(v; y. \text{with } h \text{ handle } c) \quad (\text{op} \notin \{\text{op}_1, \dots, \text{op}_n\})}$$

Fig. 3. Step relation.

Small-step operational semantics is given using a relation $c \rightsquigarrow c'$, defined in Figure 3. Observe that there is no such relation for values, as these are inert. The rules for conditionals and function application are standard. For sequencing $\text{do } x \leftarrow c_1 \text{ in } c_2$, we start by evaluating c_1 . If this returns some value v , we bind it to x and evaluate c_2 . But if c_1 calls an operation, we propagate the call outwards and defer further evaluation to the continuation of the call, as shown in Figure 4.

$$\text{do } x_1 \leftarrow (\text{do } x_2 \leftarrow \text{op}(x; y. c_2) \text{ in } c_1) \text{ in } c \rightsquigarrow \\ \text{do } x_1 \leftarrow \text{op}(x; y. \text{do } x_2 \leftarrow c_2 \text{ in } c_1) \text{ in } c \rightsquigarrow \\ \text{op}(x; y. \text{do } x_1 \leftarrow (\text{do } x_2 \leftarrow c_2 \text{ in } c_1) \text{ in } c)$$

Fig. 4. The call of op in the innermost sequencing propagates outwards until it reaches the top.

For handling **with** h **handle** c , the behaviour is similar. We start by evaluating c , and if it returns a value, we continue by evaluating the return clause of h . If c calls an operation **op**, there are two options: if h has a matching clause for **op**, we start evaluating that, passing in the parameter and the *handled* continuation; if not, we propagate the call outwards and defer further handling to the continuation, just like in sequencing.

4 Type system

To ensure that the evaluation goes smoothly, we introduce a type and effect system along the lines presented in [4,10]. Just as we split terms into values and computations, we split types into *value types* and *computation types*, given in Figure 5.

value type $A, B ::= \text{bool}$	boolean type
$A \rightarrow \underline{C}$	function type
$\underline{C} \Rightarrow \underline{D}$	handler type
computation type $\underline{C}, \underline{D} ::= A!\{\text{op}_1, \dots, \text{op}_n\}$	

Fig. 5. Syntax of types.

The value type $A \rightarrow \underline{C}$ is given to functions that take a value of type A and perform a computation of type \underline{C} , while the handler type $\underline{C} \Rightarrow \underline{D}$ is given to handlers that transform computations of type \underline{C} into ones of type \underline{D} . Every computation type has the form $A!\Delta$, where A is the type of values the computation returns, and Δ is the set of operations it *possibly* calls, i.e. the set Δ is an over-approximation of the operations that are actually called. Also note that Δ contains no information about the number of occurrences, passed parameters, or order of operations.

Typing information about operations is given in a *signature* Σ of the form

$$\{\text{op}_1 : A_1 \rightarrow B_1, \dots, \text{op}_n : A_n \rightarrow B_n\}$$

which assigns a *parameter* value type A_i and a *result* value type B_i to each operation op_i .

Example 4.1 Assuming that value types are extended with types `int` of integers, `str` of strings, `unit`, which is given to the unit value `()`, and the empty type `void`, the operations we have seen in Section 2 can be assigned the following types:

```

print : str → unit
read  : unit → str
raise : str → void
decide : unit → bool
fail  : unit → void
get   : unit → int
set   : int  → unit
    
```

Since there are no values of the void type, a call to `raise` or `fail` effectively aborts the continuation, because there are no handlers that could resume it by yielding a suitable value.

In Figure 6 we define two typing judgements: $\Gamma \vdash v : A$ for values and $\Gamma \vdash c : \underline{C}$ for computations. In both, the context Γ is an assignment of value types to variables.

$$\begin{array}{c}
 \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}} \qquad \frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \mathbf{fun } x \mapsto c : A \rightarrow \underline{C}} \\
 \\
 \frac{\left[\frac{(\mathbf{op}_i : A_i \rightarrow B_i) \in \Sigma \quad \Gamma, x : A_i, k : B_i \rightarrow B! \Delta' \vdash c_i : B! \Delta'}{\Gamma \vdash \mathbf{handler } \{ \mathbf{return } x \mapsto c_r, \mathbf{op}_1(x; k) \mapsto c_1, \dots, \mathbf{op}_n(x; k) \mapsto c_n \} : A! \Delta \Rightarrow B! \Delta'} \right]_{1 \leq i \leq n} \quad \Delta \setminus \{ \mathbf{op}_i \}_{1 \leq i \leq n} \subseteq \Delta'}{\Gamma \vdash \mathbf{handler } \{ \mathbf{return } x \mapsto c_r, \mathbf{op}_1(x; k) \mapsto c_1, \dots, \mathbf{op}_n(x; k) \mapsto c_n \} : A! \Delta \Rightarrow B! \Delta'} \\
 \\
 \frac{\Gamma \vdash v : A}{\Gamma \vdash \mathbf{return } v : A! \Delta} \qquad \frac{(\mathbf{op} : A_{\mathbf{op}} \rightarrow B_{\mathbf{op}}) \in \Sigma \quad \Gamma \vdash v : A_{\mathbf{op}} \quad \Gamma, y : B_{\mathbf{op}} \vdash c : A! \Delta \quad \mathbf{op} \in \Delta}{\Gamma \vdash \mathbf{op}(v; y.c) : A! \Delta} \\
 \\
 \frac{\Gamma \vdash c_1 : A! \Delta \quad \Gamma, x : A \vdash c_2 : B! \Delta}{\Gamma \vdash \mathbf{do } x \leftarrow c_1 \mathbf{in } c_2 : B! \Delta} \qquad \frac{\Gamma \vdash v_1 : A \rightarrow \underline{C} \quad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1 v_2 : \underline{C}} \\
 \\
 \frac{\Gamma \vdash v : \mathbf{bool} \quad \Gamma \vdash c_1 : \underline{C} \quad \Gamma \vdash c_2 : \underline{C}}{\Gamma \vdash \mathbf{if } v \mathbf{then } c_1 \mathbf{else } c_2 : \underline{C}} \qquad \frac{\Gamma \vdash v : \underline{C} \Rightarrow \underline{D} \quad \Gamma \vdash c : \underline{D}}{\Gamma \vdash \mathbf{with } v \mathbf{handle } c : \underline{D}}
 \end{array}$$

Fig. 6. Typing judgements.

Typing rules hold no surprises except for:

Return You might expect the conclusion to be $\Gamma \vdash \mathbf{return } v : A! \emptyset$ as that is the most precise type one can assign. However, we give all the rules in a form that allows coarser types because this loses no generality (e.g. in this particular rule, we can set $\Delta = \emptyset$), is sufficient for our purposes and leads to a simpler type system. See [23] for an algorithm that produces a more precise type.

Operation call Here similarly, we can assume that although Δ contains `op`, it can be assigned to the continuation c even when c does not call `op`.

Handling According to the above interpretation that $\underline{C} \Rightarrow \underline{D}$ is given to handlers that take computations of type \underline{C} to ones of type \underline{D} , it is not surprising that handling behaves like an application of a function.

Handler To give handler a type $A! \Delta \Rightarrow B! \Delta'$, we need to check that it correctly handles returned values and operations both with and without a matching operation clause. For return values, it is simple: given a value of type A , the return clause must be a computation of type $B! \Delta'$.

Next, for each handled operation $\mathbf{op}_i : A_i \rightarrow B_i$, the handling clause again needs to be a computation of type $B! \Delta'$. Here, the parameter is expected to have the type A_i as determined by Σ . Similarly, the captured continuation is a function that takes a result of type B_i and performs a computation of type $B! \Delta'$. Notice that even though the handled computation has type $A! \Delta$, the continuation has a different type because it is further handled.

Finally, we want to handle computations that call operations without a matching operation clause in the handler. For this case, we allow Δ to contain oper-

ations not in $\{\text{op}_i\}_{1 \leq i \leq n}$, but any such operation must also appear in Δ' as it may also be called in the handled computation (and thus also in continuations of handled operations).

The given typing system then ensures that well-typed computations do not get stuck [4].

Theorem 4.2 (Safety) *If $\vdash c : A! \Delta$ holds, then either:*

- $c = \text{return } v$ for some $\vdash v : A$, or
- $c = \text{op}(v; y. c')$ for some $\text{op} \in \Delta$, or
- $c \rightsquigarrow c'$ for some $\vdash c' : A! \Delta$.

5 Reasoning

Recall that two terms are *observationally equivalent* [8] if we may exchange any occurrence of the first with the second without affecting the observable properties of the surrounding program. Due to the separation in the syntax, we define observational equivalence of both computations ($c \equiv c'$) and values ($v \equiv v'$). We can show [4] that \equiv is a congruence and that it satisfies a collection of basic equivalences given in Figure 7.

$$\begin{aligned} \text{do } x \leftarrow \text{return } v \text{ in } c &\equiv c[v/x] & (1) \\ \text{do } x \leftarrow \text{op}(v; y. c_1) \text{ in } c_2 &\equiv \text{op}(v; y. \text{do } x \leftarrow c_1 \text{ in } c_2) & (2) \\ \text{do } x \leftarrow c \text{ in return } x &\equiv c & (3) \\ \text{do } x_2 \leftarrow (\text{do } x_1 \leftarrow c_1 \text{ in } c_2) \text{ in } c_3 &\equiv \text{do } x_1 \leftarrow c_1 \text{ in } (\text{do } x_2 \leftarrow c_2 \text{ in } c_3) & (4) \\ \text{if true then } c_1 \text{ else } c_2 &\equiv c_1 & (5) \\ \text{if false then } c_1 \text{ else } c_2 &\equiv c_2 & (6) \\ \text{if } v \text{ then } c[\text{true}/x] \text{ else } c[\text{false}/x] &\equiv c[v/x] & (7) \\ (\text{fun } x \mapsto c) v &\equiv c[v/x] & (8) \\ \text{fun } x \mapsto v x &\equiv v & (9) \end{aligned}$$

In the following rules, we have $h = \text{handler } \{\text{return } x \mapsto c_r, \text{op}_1(x; k) \mapsto c_1, \dots, \text{op}_n(x; k) \mapsto c_n\}$:

$$\begin{aligned} \text{with } h \text{ handle } (\text{return } v) &\equiv c_r[v/x] & (10) \\ \text{with } h \text{ handle } (\text{op}_i(v; y. c)) &\equiv c_i[v/x, (\text{fun } y \mapsto \text{with } h \text{ handle } c)/k] \quad (1 \leq i \leq n) & (11) \\ \text{with } h \text{ handle } (\text{op}(v; y. c)) &\equiv \text{op}(v; y. \text{with } h \text{ handle } c) \quad (\text{op} \notin \{\text{op}_i\}_{1 \leq i \leq n}) & (12) \\ \text{with } (\text{handler } \{\text{return } x \mapsto c_2\}) \text{ handle } c_1 &\equiv \text{do } x \leftarrow c_1 \text{ in } c_2 & (13) \end{aligned}$$

Fig. 7. Basic equivalences.

The main new tool we can use for reasoning about algebraic effects is the *induction principle* [20,4], which states that for a given predicate ϕ on computations, $\phi(c)$ holds for all computations c if:

- (i) $\phi(\text{return } v)$ holds for all values v , and
- (ii) $\phi(\text{op}(v; y. c'))$ holds for all operations op and parameters v , if we assume that $\phi(c')$ holds for all possible results y .

We can use the induction principle to derive equivalences (3), (4), and (13), but for a more interesting example, let us show that handlers *collect* and *collect'* from

Section 2.1.3 exhibit equivalent behaviour, in particular:

$$\mathbf{with\ collect\ handle\ } c \equiv \mathbf{do\ } g \leftarrow (\mathbf{with\ collect' \ handle\ } c) \mathbf{in\ } g \text{ “”}$$

To succeed with induction, we need to prove a stronger statement that for any string s_0 , we have

$$\begin{aligned} & \mathbf{do\ } (x_1, s_1) \leftarrow (\mathbf{with\ collect\ handle\ } c) \mathbf{in\ return\ } (x_1, \mathbf{join\ } s_0\ s_1) \equiv \\ & \mathbf{do\ } g \leftarrow (\mathbf{with\ collect' \ handle\ } c) \mathbf{in\ } g\ s_0 \end{aligned}$$

We recover the desired goal by setting $s_0 = \text{“”}$. The induction on c goes as follows:

- (i) The base case is trivial: if $c = \mathbf{return\ } v$, both sides are equal to $\mathbf{return\ } (v, s_0)$.
- (ii) For the induction step when $c = \mathbf{op}(v; y. c')$, we have two possibilities: either $\mathbf{op} \neq \mathbf{print}$, which is again trivial, or $\mathbf{op} = \mathbf{print}$, where we show:

$$\begin{aligned} & \mathbf{do\ } (x_1, s_1) \leftarrow (\mathbf{with\ collect\ handle\ print}(s_2; _ . c')) \mathbf{in\ return\ } (x_1, \mathbf{join\ } s_0\ s_1) \\ & \equiv (11) \ \& \ (8) \\ & \mathbf{do\ } (x_1, s_1) \leftarrow (\\ & \quad \mathbf{do\ } (x, acc) \leftarrow (\mathbf{with\ collect\ handle\ } c') \mathbf{in\ return\ } (x, \mathbf{join\ } s_2\ acc) \\ & \quad) \mathbf{in\ return\ } (x_1, \mathbf{join\ } s_0\ s_1) \\ & \equiv (4) \\ & \mathbf{do\ } (x, acc) \leftarrow (\mathbf{with\ collect\ handle\ } c') \mathbf{in} \\ & \mathbf{do\ } (x_1, s_1) \leftarrow (\mathbf{return\ } (x, \mathbf{join\ } s_2\ acc)) \mathbf{in} \\ & \mathbf{return\ } (x_1, \mathbf{join\ } s_0\ s_1) \\ & \equiv (1) \\ & \mathbf{do\ } (x, acc) \leftarrow (\mathbf{with\ collect\ handle\ } c') \mathbf{in\ return\ } (x, \mathbf{join\ } s_0\ (\mathbf{join\ } s_2\ acc)) \\ & \equiv (\text{associativity of join}) \\ & \mathbf{do\ } (x, acc) \leftarrow (\mathbf{with\ collect\ handle\ } c') \mathbf{in\ return\ } (x, \mathbf{join\ } (\mathbf{join\ } s_0\ s_2)\ acc) \\ & \equiv (\text{induction hypothesis}) \\ & \mathbf{do\ } f \leftarrow (\mathbf{with\ collect' \ handle\ } c') \mathbf{in\ } f\ (\mathbf{join\ } s_0\ s_2) \\ & \equiv (1) \ \& \ (8) \\ & \mathbf{do\ } g \leftarrow \mathbf{return\ } (\\ & \quad \mathbf{fun\ } acc \mapsto \mathbf{do\ } f \leftarrow (\mathbf{with\ collect' \ handle\ } c') \mathbf{in\ } f\ (\mathbf{join\ } acc\ s_2) \\ & \quad) \mathbf{in\ } g\ s_0 \\ & \equiv (11) \ \& \ (8) \\ & \mathbf{do\ } g \leftarrow (\mathbf{with\ collect' \ handle\ print}(s_2; _ . c')) \mathbf{in\ } g\ s_0 \end{aligned}$$

6 Further reading

6.1 Call-by-push-value

Call-by-push-value [12] is an evolved version of the fine-grain call-by-value approach. Though the latter was used in this tutorial as it is closer to the more familiar call-by-value, a significant part of the recent work on algebraic effects uses the former.

To compare given operational semantics and effect system to ones done in a call-by-push-value setting, see [10], while for denotational semantics and reasoning, see [22].

6.2 Programming with handlers

The list of examples in Section 2 is by no means exhaustive. For more involved examples that include multi-threading, delimited continuations, selection functionals, text processing, resource management, efficient backtracking, or logic programming, see [5,10,6,25]. A number of implementations of handlers has also sprung up, either as independent languages [3,14], or as libraries in existing languages [10,6,25]. More recently, a multicore [2] branch of OCaml [1] has started adopting handlers as a way of implementing concurrency primitives.

6.3 Denotational semantics

In the naive setting where operations return only first-order values and there is no recursion, we can interpret each value type A with a set $\llbracket A \rrbracket$, while a computation type $\llbracket A! \Delta \rrbracket$ is interpreted as the set of trees (like ones described in Section 1) with leaves in $\llbracket A \rrbracket$ and nodes corresponding to operations in Δ . Handlers are interpreted as functions between trees, and are defined by structural recursion on the tree of the handled computation, while handling is interpreted by application of such functions.

More abstractly, we define a *model* of Δ to be a set M together with a map $\text{op}_M: \llbracket A \rrbracket \times M^{\llbracket B \rrbracket} \rightarrow M$ for each operation $\text{op}: A \rightarrow B \in \Delta$, while a *homomorphism* between models M and N is defined to be a map $h: M \rightarrow N$ such that $(h \circ \text{op}_M)(x, k) = \text{op}_N(x, h \circ k)$. It turns out that $\llbracket A! \Delta \rrbracket$ is exactly the *free model* of Δ over $\llbracket A \rrbracket$, i.e. a model characterized with the following universal property: given any model M of Δ and any map $f: \llbracket A \rrbracket \rightarrow M$, there exists a unique homomorphism $h: \llbracket A! \Delta \rrbracket \rightarrow M$ that agrees with f on leaves. We can use this universal property to interpret handlers: operation clauses define a model of operations, and the return clause provides a function f that can be extended to a homomorphism.

For more detail, see [22]. In the general setting with recursion and higher-order results, we need to switch from sets to domains, but the general idea is the same [4].

6.4 Algebraic theories

Traditionally, algebraic effects were described not only by a set of operations, but also by an equational theory that captures their properties. For example, nondeterminism can be represented with a binary operation `decide` and equations stating its idempotency, commutativity, and associativity [18,9,17]. The benefit of equations is that they validate certain program optimizations [11] and better capture the effectful behaviour of operations. With various extensions of such theories, one can also describe complicated effects such as control-flow jumps [7] even in the absence of handlers, or quantum computation [24].

However, a lot of computationally interesting handlers (for example `backtrack` from Section 2.3.2) do not respect these equations and thus cannot receive a homomorphic interpretation described above [22]. For this reason, current research on handlers assumes no such equations, but connections exists in both directions:

on one hand, we can still apply previous results by assuming a trivial equational theory, and on the other hand, we can use reasoning techniques to recover equations from the behaviour of handlers [4].

6.5 Modelling actual effects

One can model “real-world” effects with a *comodel*, which is a set W representing the possible world states together with a map $\text{op}^W : W \times \llbracket A \rrbracket \rightarrow W \times \llbracket B \rrbracket$ for each operation $\text{op} : A \rightarrow B \in \Sigma$. Thus, when an operation call $\text{op}(v; y. c)$ escapes all handlers, we pass the current state $w \in W$ and the parameter v to op^W and get back the new state and a result, which we assign to y and continue evaluating c . For more details, see [5, Section 4.1], which is based on a more abstract treatment in [19], where the duality between models and comodels is explained in more detail.

Acknowledgement

I want to thank Andrej Bauer and Alex Simpson for their truly helpful feedback.

References

- [1] *Ocaml*.
URL <http://ocaml.org>
- [2] *Ocaml multicore branch*.
URL <https://github.com/ocaml-labs/ocaml-multicore>
- [3] Bauer, A. and M. Pretnar, *Eff*.
URL <http://www.eff-lang.org>
- [4] Bauer, A. and M. Pretnar, *An effect system for algebraic effects and handlers*, Logical Methods in Computer Science **10** (2014).
URL [http://dx.doi.org/10.2168/LMCS-10\(4:9\)2014](http://dx.doi.org/10.2168/LMCS-10(4:9)2014)
- [5] Bauer, A. and M. Pretnar, *Programming with algebraic effects and handlers*, J. Log. Algebr. Meth. Program. **84** (2015), pp. 108–123.
URL <http://dx.doi.org/10.1016/j.jlamp.2014.02.001>
- [6] Brady, E., *Programming and reasoning with algebraic effects and dependent types*, in: G. Morrisett and T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP’13, Boston, MA, USA - September 25 - 27, 2013* (2013), pp. 133–144.
URL <http://doi.acm.org/10.1145/2500365.2500581>
- [7] Fiore, M. P. and S. Staton, *Substitution, jumps, and algebraic effects*, in: T. A. Henzinger and D. Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS ’14, Vienna, Austria, July 14 - 18, 2014* (2014), p. 41.
URL <http://doi.acm.org/10.1145/2603088.2603163>
- [8] Harper, R., “Practical Foundations for Programming Languages,” Cambridge University Press, 2012.
- [9] Hyland, M., G. D. Plotkin and J. Power, *Combining effects: Sum and tensor*, Theor. Comput. Sci. **357** (2006), pp. 70–99.
URL <http://dx.doi.org/10.1016/j.tcs.2006.03.013>
- [10] Kammar, O., S. Lindley and N. Oury, *Handlers in action*, in: G. Morrisett and T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP’13, Boston, MA, USA - September 25 - 27, 2013* (2013), pp. 145–158.
URL <http://doi.acm.org/10.1145/2500365.2500590>
- [11] Kammar, O. and G. D. Plotkin, *Algebraic foundations for effect-dependent optimisations*, in: J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012* (2012), pp. 349–360.
URL <http://doi.acm.org/10.1145/2103656.2103698>

- [12] Levy, P. B., “Call-By-Push-Value: A Functional/Imperative Synthesis,” *Semantics Structures in Computation* **2**, Springer, 2004.
- [13] Levy, P. B., J. Power and H. Thielecke, *Modelling environments in call-by-value programming languages*, *Inf. Comput.* **185** (2003), pp. 182–210.
URL [http://dx.doi.org/10.1016/S0890-5401\(03\)00088-9](http://dx.doi.org/10.1016/S0890-5401(03)00088-9)
- [14] McBride, C., *Frank*.
URL <https://hackage.haskell.org/package/Frank/>
- [15] Pierce, B. C., “Types and programming languages,” MIT Press, 2002.
- [16] Plotkin, G. D. and J. Power, *Adequacy for algebraic effects*, in: F. Honsell and M. Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, Lecture Notes in Computer Science **2030** (2001), pp. 1–24.
URL http://dx.doi.org/10.1007/3-540-45315-6_1
- [17] Plotkin, G. D. and J. Power, *Notions of computation determine monads*, in: M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, Lecture Notes in Computer Science **2303** (2002), pp. 342–356.
URL http://dx.doi.org/10.1007/3-540-45931-6_24
- [18] Plotkin, G. D. and J. Power, *Algebraic operations and generic effects*, *Applied Categorical Structures* **11** (2003), pp. 69–94.
URL <http://dx.doi.org/10.1023/A:1023064908962>
- [19] Plotkin, G. D. and J. Power, *Tensors of comodels and models for operational semantics*, *Electr. Notes Theor. Comput. Sci.* **218** (2008), pp. 295–311.
URL <http://dx.doi.org/10.1016/j.entcs.2008.10.018>
- [20] Plotkin, G. D. and M. Pretnar, *A logic for algebraic effects*, in: *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA* (2008), pp. 118–129.
URL <http://dx.doi.org/10.1109/LICS.2008.45>
- [21] Plotkin, G. D. and M. Pretnar, *Handlers of algebraic effects*, in: G. Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, Lecture Notes in Computer Science **5502** (2009), pp. 80–94.
URL http://dx.doi.org/10.1007/978-3-642-00590-9_7
- [22] Plotkin, G. D. and M. Pretnar, *Handling algebraic effects*, *Logical Methods in Computer Science* **9** (2013).
URL [http://dx.doi.org/10.2168/LMCS-9\(4:23\)2013](http://dx.doi.org/10.2168/LMCS-9(4:23)2013)
- [23] Pretnar, M., *Inferring algebraic effects*, *Logical Methods in Computer Science* **10** (2014).
URL [http://dx.doi.org/10.2168/LMCS-10\(3:21\)2014](http://dx.doi.org/10.2168/LMCS-10(3:21)2014)
- [24] Staton, S., *Algebraic effects, linearity, and quantum programming languages*, in: S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015* (2015), pp. 395–406.
URL <http://doi.acm.org/10.1145/2676726.2676999>
- [25] Wu, N., T. Schrijvers and R. Hinze, *Effect handlers in scope*, in: W. Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014* (2014), pp. 1–12.
URL <http://doi.acm.org/10.1145/2633357.2633358>