An Introduction to
Algebraic Effects and Handlers
Invited tutorial paper

Matija Pretnar
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,
value $v ::= x$| variable
| | true | false | boolean constants
| | $\text{fun } x \mapsto c$ | function
| | $h$ | handler

handler $h ::= \text{handler} \{ \text{return } x \mapsto c_v, \}\quad (\text{optional) return clause}$

operation clauses

computation $c ::= \text{return } v$| return
| $\text{op}(v; y. c)$ | operation call
| $\text{do } x \leftarrow c_1 \text{ in } c_2$ | sequencing
| $\text{if } v \text{ then } c_1 \text{ else } c_2$ | conditional
| $v_1 v_2$ | application
| $\text{with } v \text{ handle } c$ | handling

Fig. 1. Syntax of terms.

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

**Sequencing** In $\text{do } x \leftarrow c_1 \text{ in } c_2$, 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 $\text{op}(v; y. c)$ passes a parameter value $v$ (e.g. the memory location to be read) to the operation $\text{op}$, and after $\text{op}$ 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 $\text{op}$, which takes a parameter and passes it to an operation call with the trivial continuation:

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

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

**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 $\text{rec fun } f x \mapsto c$, 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 $(\text{fun } (x, _) \mapsto 6 + x) (7, 8)$.

**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:

\[
\text{(fun } (x,\_)) \mapsto \text{do } f \leftarrow (+) 6 \text{ in } f x) (7, 8)
\]

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 `fun x \mapsto \text{fun } y \mapsto (x, y)`
instead of `fun x \mapsto \text{return } \text{(fun } y \mapsto \text{return } (x, y))`.

**Semantics** Observe that each operation call creates a branching point in the eval-
uation, 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.](image)

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:

\[
\text{printFullName} \equiv \text{print} \text{"What is your forename?";} \\
\text{do forename} \leftarrow \text{read}() \text{ in} \\
\text{print} \text{"What is your surname?";} \\
\text{do surname} \leftarrow \text{read}() \text{ in} \\
\text{print} (\text{join forename surname})
\]

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:

\[
\text{handler} \{ \text{read}(_; k) \mapsto k \text{ "Bob" } \}
\]

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:

\[
\text{alwaysRead} \equiv \text{fun } s \mapsto \text{handler} \{ \text{read}(_; k) \mapsto k s \}
\]

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

\[
\text{with (alwaysRead "Bob") handle printFullName}
\]

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 \text{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

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

then \textbf{with reverse handle abc} prints out first \text{"C"}, then \text{"B"}, and finally \text{"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:

\[
\text{collect} \stackrel{\text{def}}{=} \text{handler} \{ \text{return } x \mapsto \text{return } (x, \text{""})
\]

\[
\text{print}(s; k) \mapsto \\
\text{do } (x, \text{acc}) \leftarrow k() \text{ in}
\]

\[
\text{return } (x, \text{join } \text{acc})\}
\]

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 \(\text{acc}\) in addition to the final value \(x\). Now, we only need to join \(s\) with \(\text{acc}\) and return it together with \(x\). If we handle \text{abc} with \text{collect}, we get a pair ((), “A B C”), where () is the unit result of the last print.

We can also nest handlers, and

\textbf{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 \text{collect} handles all print calls, and so none reach the \text{reverse} handler, which then does nothing.

Alternatively, we could implement the same handler using a technique called \textit{parameter-passing} \cite{22}, where we transform the handled computation into a function that passes around a parameter, in our case the accumulated string:

\[
\text{collect}’ \stackrel{\text{def}}{=} \text{handler} \{ \text{return } x \mapsto \text{fun } \text{acc} \mapsto \text{return } (x, \text{acc})
\]

\[
\text{print}(s; k) \mapsto \\
\text{fun } \text{acc} \mapsto (k()) (\text{join } \text{acc } s)\}
\]
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:

\[
\text{(with collect' handle } c \text{'')}\]

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 \( \text{raise} \) 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:

\[
def \text{default} = \text{fun } x \mapsto \text{handler} \{ \text{raise} (_) \mapsto \text{return} x \}
\]

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 \( \text{decide} \), which takes a unit parameter, and non-deterministically yields a boolean. Then, a binary choice can be implemented as a function

\[
def \text{choose} = \text{fun } (x, y) \mapsto \\
\text{do } b \leftarrow \text{decide}() \text{ in} \\
\text{if } b \text{ then } (\text{return } x) \text{ else } (\text{return } y)
\]

However, unlike \( print \), we assume no intrinsic behaviour for \( \text{decide} \), 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 \( \text{choose}(3, 4) \) is stuck when it encounters the \( \text{decide} \) call. The simplest handler for \( \text{decide} \) is

\[
def \text{pickTrue} = \text{handler} \{ \text{decide}(_, k) \mapsto k \text{ true} \}
\]
which makes each decide yield true to the continuation, so choose always chooses
the left argument. So, if we define

\[
\text{chooseDiff} \overset{\text{def}}{=} \text{do } x_1 \leftarrow \text{choose}(15, 30) \text{ in } \\
\text{do } x_2 \leftarrow \text{choose}(5, 10) \text{ in } \\
\text{return } (x_1 - x_2)
\]

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:

\[
\text{pickMax} \overset{\text{def}}{=} \text{handler } \{ \text{decide}_<(k) \mapsto } \\
\text{do } x_t \leftarrow k \text{ true in } \\
\text{do } x_f \leftarrow k \text{ false in } \\
\text{return } \max(x_t, x_f) \}
\]

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:

\[
\text{rec fun } \text{chooseInt}(m, n) \mapsto \\
\text{if } m > n \text{ then fail }() \text{ else } \\
\text{do } b \leftarrow \text{decide }() \text{ in } \\
\text{if } b \text{ then (return } m) \text{ else chooseInt}(m + 1, n)
\]

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

\[
\text{pythagorean} \overset{\text{def}}{=} \text{fun } (m, n) \mapsto \\
\text{do } a \leftarrow \text{chooseInt}(m, n - 1) \text{ in } \\
\text{do } b \leftarrow \text{chooseInt}(a + 1, n) \text{ in } \\
\text{if isSquare } (a^2 + b^2) \text{ then (return } (a, b, \sqrt{a^2 + b^2})) \text{ else fail }()
\]
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`:

\[
\text{backtrack} \overset{\text{def}}{=} \text{handler}\{\text{decide}(\_;k) \mapsto \\
\text{with} \\
\text{handler}\{\text{fail}(\_;\_ ) \mapsto k \text{ false}\}
\text{handle} \\
k \text{ true}\}
\]

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:

\[
\text{state} \overset{\text{def}}{=} \text{handler}\{\text{get}(\_;k) \mapsto \text{fun} s \mapsto (k \ s) \ s \\
\text{set}(s;k) \mapsto \text{fun} \_ \mapsto (k () ) \ s \\
\text{return} \ x \mapsto \text{fun} \_ \mapsto \text{return} \ x\}
\]

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:

\[
\text{transaction} \overset{\text{def}}{=} \text{handler} \{ \text{get}(\_; k) \mapsto \text{fun} \ s \mapsto (k \ s) \ s \\
\text{set}(s; k) \mapsto \text{fun} \ _\mapsto (k \ ()) \ s \\
\text{return} \ x \mapsto \text{fun} \ s \mapsto \text{set} \ s; \text{return} \ x \} 
\]

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.

\[
\begin{array}{ll}
\text{do } x \leftarrow c_1 \text{ in } c_2 \leadsto \text{do } x \leftarrow c'_1 \text{ in } c_2 & \text{do } x \leftarrow \text{return} \ v \text{ in } c \leadsto c[v/x] \\
\text{if false then } c_1 \text{ else } c_2 \leadsto c_2 & \text{if true then } c_1 \text{ else } c_2 \leadsto c_1 \\
\text{do } x \leftarrow \text{op}(v; y, c_1) \text{ in } c_2 \leadsto \text{op}(v; y, \text{do } x \leftarrow c_1 \text{ in } c_2) & \text{(fun } x \mapsto c) \ v \leadsto c[v/x] \\
\text{with } h \text{ handle } c \leadsto \text{with } h \text{ handle } c' & \text{with } h \text{ handle } (\text{return } v) \leadsto c[v/x] \\
\text{with } h \text{ handle } \text{op}_i(v; y, c) \leadsto c[v/x], (\text{fun } y \mapsto \text{with } h \text{ handle } c)/k & (1 \leq i \leq n) \\
\text{with } h \text{ handle } \text{op}(v; y, c) \leadsto \text{op}(v; y, \text{with } h \text{ handle } c) & (\text{op} \notin (\text{op}_1, \ldots, \text{op}_n))
\end{array}
\]

Fig. 3. Step relation.

Small-step operational semantics is given using a relation \( c \leadsto c' \), 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.

\[
\begin{array}{l}
\text{do } x_1 \leftarrow (\text{do } x_2 \leftarrow \text{op}(x; y, c_2) \text{ in } c_1) \text{ in } c \leadsto \\
\text{do } x_1 \leftarrow \text{op}(x; y, \text{do } x_2 \leftarrow c_2 \text{ in } c_1) \text{ in } c \leadsto \\
\text{op}(x; y, \text{do } x_1 \leftarrow (\text{do } x_2 \leftarrow c_2 \text{ in } c_1) \text{ in } c) 
\end{array}
\]

Fig. 4. The call of \text{op} in the innermost sequencing propagates outwards until it reaches the top.
For handling with \texttt{h} handle \texttt{c}, the behaviour is similar. We start by evaluating \texttt{c}, and if it returns a value, we continue by evaluating the return clause of \texttt{h}. If \texttt{c} calls an operation \texttt{op}, there are two options: if \texttt{h} has a matching clause for \texttt{op}, we start evaluating that, passing in the parameter and the \texttt{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 \textit{value types} and \textit{computation types}, given in Figure 5.

\begin{align*}
\text{value type } A, B &::= \text{bool} \\
&| A \rightarrow C \\
&| C \Rightarrow D \\
\text{computation type } C, D &::= A!\{\text{op}_1, \ldots, \text{op}_n\}
\end{align*}

Fig. 5. Syntax of types.

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

Typing information about operations is given in a \textit{signature} \(\Sigma\) of the form

\[\{\text{op}_1 : A_1 \rightarrow B_1, \ldots, \text{op}_n : A_n \rightarrow B_n\}\]

which assigns a \textit{parameter} value type \(A_i\) and a \textit{result} value type \(B_i\) to each operation \(\text{op}_i\).

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

- \texttt{print : str \rightarrow unit}
- \texttt{read : unit \rightarrow str}
- \texttt{raise : str \rightarrow void}
- \texttt{decide : unit \rightarrow bool}
- \texttt{fail : unit \rightarrow void}
- \texttt{get : unit \rightarrow int}
- \texttt{set : int \rightarrow 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 \( \Gamma \vdash c : C \) for computations. In both, the context \( \Gamma \) is a assignment of value types to variables.

<table>
<thead>
<tr>
<th>((x : A) \in \Gamma)</th>
<th>(\Gamma \vdash x : A)</th>
<th>(\Gamma \vdash \text{true} : \text{bool})</th>
<th>(\Gamma \vdash \text{false} : \text{bool})</th>
<th>(\Gamma, x : A \vdash c : C)</th>
<th>(\Gamma \vdash \text{fun} \ x \mapsto c : A \rightarrow C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>({\text{op}_1 : A_1 \rightarrow B_1} \in \Sigma)</td>
<td>(\Gamma, x : A_i, k : B_i \rightarrow B!\Delta' + c_i : B!\Delta'_{1 \leq i \leq n})</td>
<td>(\Delta \setminus {\text{op}<em>i}</em>{1 \leq i \leq n} \subseteq \Delta')</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\Gamma \vdash \text{handler} \ (\text{return} \ x \mapsto c_r, \text{op}_1(x;k) \mapsto c_1, \ldots, \text{op}_n(x;k) \mapsto c_n) : A!\Delta \Rightarrow B!\Delta')</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(\Gamma \vdash v : A)</th>
<th>({\text{op} : A_{\text{op}} \rightarrow B_{\text{op}}} \in \Sigma)</th>
<th>(\Gamma \vdash v : A_{\text{op}})</th>
<th>(\Gamma, y : B_{\text{op}} \vdash c : A!\Delta)</th>
<th>(\text{op} \in \Delta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Gamma \vdash c_1 : A!\Delta)</td>
<td>(\Gamma, x : A \vdash c_2 : B!\Delta)</td>
<td>(\Gamma \vdash \text{do} \ \ x \leftarrow c_1 \ \ \text{in} \ c_2 : B!\Delta)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\Gamma \vdash v : \text{bool})</td>
<td>(\Gamma \vdash c_1 : C)</td>
<td>(\Gamma \vdash c_2 : C)</td>
<td>(\Gamma \vdash \text{if} \ v \ \text{then} \ c_1 \ \text{else} \ c_2 : C)</td>
<td></td>
</tr>
<tr>
<td>(\Gamma \vdash v \ : C \Rightarrow D)</td>
<td>(\Gamma \vdash v \ : C)</td>
<td>(\Gamma \vdash c : C)</td>
<td>(\Gamma \vdash \text{with} \ v \ \text{handle} \ c : D)</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 6. Typing judgements.

Typing rules hold no surprises except for:

**Return** You might expect the conclusion to be \( \Gamma \vdash \text{return} \ v : A!\emptyset \) 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 \( \Delta \) contains \( \text{op} \), it can be assigned to the continuation \( c \) even when \( c \) does not call \( \text{op} \).

**Handling** According to the above interpretation that \( C \Rightarrow D \) is given to handlers that take computations of type \( C \) to ones of type \( D \), 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 \( \text{op}_i : A_i \rightarrow B_i \), 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 \( \Sigma \). 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 \( \Delta \) to contain oper-
for a more interesting example, let us show that handlers collect.

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
- \( c \leadsto c' \) 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.

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

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

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

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 \( \phi \) 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 \( \text{op} \) 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:

\[
\text{with collect handle } c \equiv \text{do } g \leftarrow (\text{with collect' handle } c) \text{ in } g \
\]

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

\[
\text{do } (x_1, s_1) \leftarrow (\text{with collect handle } c) \text{ in return } (x_1, \text{join } s_0 s_1) \equiv \\
\text{do } g \leftarrow (\text{with collect' handle } c) \text{ in } g s_0
\]

We recover the desired goal by setting \( s_0 = "\)". The induction on \( c \) goes as follows:

(i) The base case is trivial: if \( c = \text{return } v \), both sides are equal to \( \text{return } (v, s_0) \).

(ii) For the induction step when \( c = \text{op}(v; y, c' \) ), we have two possibilities: either \( \text{op} \neq \text{print} \), which is again trivial, or \( \text{op} = \text{print} \), where we show:

\[
\text{do } (x_1, s_1) \leftarrow (\text{with collect handle print}(s_2; \ldots, c')) \text{ in return } (x_1, \text{join } s_0 s_1) \\
\equiv (11) \& (8)
\]

\[
\text{do } (x, acc) \leftarrow (\text{with collect handle } c') \text{ in return } (x, \text{join } s_2 acc) \\
\rightarrow \text{do } (x_1, s_1) \leftarrow (\text{return } (x, \text{join } s_0 s_1)) \text{ in return } (x_1, \text{join } s_0 s_1) \\
\equiv (4)
\]

\[
\text{do } f \leftarrow (\text{with collect' handle } c') \text{ in } f (\text{join } s_0 s_2) \\
\equiv (1) \& (8)
\]

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 $\Delta$. 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 $\Delta$ to be a set $M$ together with a map $\text{op}_M : \llbracket A \rrbracket \times M^{[\Delta]} \to M$ for each operation $\text{op} : A \to B \in \Delta$, while a homomorphism between models $M$ and $N$ is defined to be a map $h : M \to N$ 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 $\Delta$ over $\llbracket A \rrbracket$, i.e. a model characterized with the following universal property: given any model $M$ of $\Delta$ and any map $f : \llbracket A \rrbracket \to M$, there exists a unique homomorphism $h : \llbracket A! \Delta \rrbracket \to M$ 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 $\text{decide}$ 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 \( \op^W : W \times \llbracket A \rrbracket \rightarrow W \times \llbracket B \rrbracket \) for each operation \( \op : A \rightarrow B \in \Sigma \). Thus, when an operation call \( \op(v; y, c) \) 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

URL http://ocaml.org

URL https://github.com/ocaml/ocaml-multicore

URL http://www.eff-lang.org

URL http://dx.doi.org/10.2168/LMCS-10(4:9)2014

URL http://dx.doi.org/10.1016/j.jlamp.2014.02.001

URL http://doi.acm.org/10.1145/2500368.2500981

URL http://doi.acm.org/10.1145/2603088.2603163


URL http://dx.doi.org/10.1016/j.tcs.2006.03.013

URL http://doi.acm.org/10.1145/2500368.2500990

URL http://doi.acm.org/10.1145/2103656.2103698


