One-Shot Algebraic Effects as Coroutines

Satoru Kawahara and Yukiyoshi Kameyama(✉)

Department of Computer Science, University of Tsukuba, Tsukuba, Japan
sat@logic.cs.tsukuba.ac.jp, kameyama@acm.org

Abstract. This paper presents a translation from algebraic effects and handlers to asymmetric coroutines, which provides a simple, efficient and widely applicable implementation for the former. Algebraic effects and handlers are emerging as main-stream language technology to model effectful computations and attract attention not only from researchers but also from programmers. They are implemented in various ways as part of compilers, interpreters, or as libraries. We present a direct embedding of one-shot algebraic effects and handlers in a language which has asymmetric coroutines. The key observation is that, by restricting the use of continuations to be *one-shot*, we obtain a simple and sufficiently general implementation via coroutines, which are available in many modern programming languages. Our translation is a macro-expressible translation, and we have implemented its embedding as a library in Lua and Ruby, which allows one to write effectful programs in a modular way using algebraic effects and handlers.

Keywords: Algebraic effect and handler · Coroutine · Continuation · Control operator · Macro expressibility

1 Introduction

Algebraic effects [21] and handlers [22] (AEH for short) are emerging as main-stream language technology to model effectful computations in a modular way. They are gaining more and more attention not only from researchers but also from practitioners. There are a few dedicated programming languages such as Eff [1], Multicore OCaml [7] and Koka [17] which have AEH as language primitives, and several main-stream programming languages such as Haskell, OCaml, Scala, JVM bytecode and C have library implementations for AEH. However, AEH is not yet available in many other main-stream programming languages, which is a big obstacle to utilize theoretical results on AEH in real-world software. We, therefore, think that it is an important and timely issue to develop a systematic and efficient implementation method for AEH which is available in many existing programming languages.

AEH have been so far implemented in several ways such as the one based on stack manipulation, delimited-control operators [6], or free monad. Unfortunately, none of them are fully satisfactory; The implementation method based on stack manipulation is used for JVM bytecode and C [4,18], however, an

© Springer Nature Switzerland AG 2020
A. Byrski and J. Hughes (Eds.): TFP 2020, LNCS 12222, pp. 159–179, 2020.
https://doi.org/10.1007/978-3-030-57761-2_8

implementer needs deep insight on the internal structure of run-time systems. It then follows that the implementation cost is rather high, which prevents the feature from being implemented in various language systems. The implementation method via delimited-control operators is used for OCaml and Scala [3,16]. It is a systematic way to implement AEH, since it needs no knowledge on low-level features, however, only few languages have delimited-control operators as built-in primitives. The implementation method based on free monads is yet another systematic way, and used in Haskell and Scala [14,15]. While elegant, it enforces a programmer to use the monadic style, and it is often inefficient.

This paper presents a new systematic method of implementing algebraic effects and handlers which is simple, efficient, and available in many languages. compared to the existing implementations based on free monads. The key of our method is to use coroutines to embed them in programming languages. Today we see a number of programming languages which have coroutines as a built-in feature.[1] which makes it possible to apply our implementation method in various languages with no or little cost. While coroutines are less expressive than general delimited-control operators, they are as expressive as *one-shot* delimited control-operators, a restricted control operator that is allowed to invoke a delimited continuation at most once [20]. One-shot delimited-control operators are known to be implemented more efficiently than general, multi-shot ones, thanks to the fact that no copying of continuations is necessary [5]. Hence, we face the trade-off between expressiveness and efficiency. This paper studies the one-shot variant which gives less expressive, but more performant primitives for AEH. In fact, various control effects are expressible by the one-shot variant.

We translate one-shot AEH to asymmetric coroutines. The salient feature of our translation is that it is a *macro-expressible translation* in the sense of Felleisen [8]. Thanks to this property, we can implement AEH as a simple library, and we have built AEH libraries for Lua and Ruby which have been published via GitHub[2,3]. Our libraries have been used by several users, and interested users have ported our libraries to other languages such as JavaScript and Rust using generators[4,5].

Our main contributions in this paper are the following.

- We show an embedding of one-shot algebraic effects and handlers. We use standard asymmetric coroutines only, and no special control features are needed. Hence our embedding is applicable to various languages as long as they have asymmetric coroutines.
- Comparing to the embedding based on free monads, our method does not force programmers to use monadic style, and our embedding is more performant in many cases than the one based on free monads.

[1] The Wikipedia article on coroutine (access date: June 1, 2020) lists fifty programming languages which have native support for coroutines.

[2] https://github.com/nymphium/eff.lua.

[3] https://github.com/nymphium/ruff.

[4] https://github.com/MakeNowJust/eff.js.

[5] https://github.com/pandaman64/effective-rust.

– Our embedding is defined as a local and compositional translation from algebraic effects and handlers. Thanks to this property, we can implement the embedding as a library, and in fact we have done it for Lua and Ruby, which is available on GitHub. Implementing AEH is a complicated task which is often error prone, and our simplistic approach based on a formal translation is desirable.

This paper is organized as follows. Section 2 shows typical examples using AEH and demonstrates our algebraic-effect library for Lua. Section 3 describes the embedding method by defining the translation from λ_{eff}, a language with algebraic effect handlers, to λ_{ac}, a language with asymmetric coroutines. We also show that our translation is macro expressible in the sense of Felleisen. Section 4 discusses the extension of our model definitions for implementing our libraries in Lua and Ruby, and problems in actual use. Section 5 shows the performance evaluation of our embedding by comparing ours with the embedding based on free monads. Section 6 describes related work, and Sect. 7 concludes.

2 Examples of *One-Shot* Algebraic Effects

This section illustrates programming with AEH by examples. To express them, we use the programming language Lua extended with our library, which is implemented using our embedding explained in the subsequent sections.

2.1 Exception

In our view, AEH is a generalization of exceptions, which is justified by the following examples.

The function `inst` provided by our library creates, when called with zero argument, a new label for an algebraic effect, and returns it.

```
1 local DivideByZero = inst()
```

We can invoke the labeled effect by calling the function `perform` in our library.

```
1 local div = function(x, y)
2   if y == 0 then
3     return perform(DivideByZero, nil)
4   else
5     return x / y
6   end
7 end
```

This code snippet is Lua's definition for the function `div`, which takes two arguments `x` and `y`. It returns the result of dividing `x` by `y` unless `y` is `0`. If `y` is `0`, it performs the effect labeled by `DivideByZero`, which means that an effect is raised and the control of the program is brought to the nearest effect handler (which is not shown in the above code) similarly to exception handling.

Our library has the function `handler` which creates a new effect handler.

```
1 local with_nil = handler {
2   val = function(_) return nil end,
3   [DivideByZero] = function(_, _)
4     return nil
5   end
6 }
```

The function `handler` receives a `table`, Lua's data structure for an associative array[6], as its sole argument. On Line 2, `e1 = e2` represents the key-value pair `["e1"] = e2`, which has the key `val` and whose value is a value handler which is used when no effect occurs. The second key-value pair (Lines 3 and 4) defines how the effect `DivideByZero` is processed. The value part of the key-value pair is a function in both cases. While the value handler receives one argument (which corresponds to the result of the handled expression), the effect handler receives two arguments, the first of which is the argument of the effect invocation and the second is a delimited continuation when the effect has been invoked (up to the handler invocation). In the above snippet, the arguments are ignored, and the whole computation returns `nil` in both cases, representing simple exception capturing. By evaluating `with_nil(function() return div(3, 0) end)`, we get `nil` as the result.

We can turn the above simple exception to a *resumable* exception by changing the effect handler as follows.

```
1 local with_default_zero = handler {
2   val = function(v) return v end,
3   [DivideByZero] = function(_, k)
4     return k(0)
5   end
6 }
```

Here we changed the second case of the handler (Lines 3 and 4) so that a parameter `k` is bound to the second argument (a delimited continuation), which is invoked with the argument `0`, and its value becomes the final result.

We can test the handler `with_default_zero` as follows.

```
1 with_default_zero(function()
2   local v = div(3, 0)
3   return v + 20
4 end)
```

[6] https://www.lua.org/manual/5.3/manual.html#3.4.9.

When we execute Line 2 of this code, the effect `DivideByZero` is performed (raised) as before. Then the handler `with_default_zero` catches it, and captures the delimited continuation `local v = □; return v + 20`, to which the variable `k` is bound. (Strictly speaking, the delimited continuation should be surrounded by the handler `with_default_zero`, but we omit it here since there is no effect in the continuation and its value handler is the identity function.) Then we execute `k(0)`, which is equivalent to `local v = 0; return v + 20`. The net effect is the same as the case when `div(3,0)` returns `0`, and the entire computation results in `0 + 20 = 20`.

2.2 State

AEH can express not only exceptions, but also many other effects. Here, we show how state can be expressed in terms of these operations using the state-passing technique.

We first create two effect labels.

```
1 local Get = inst()
2 local Put = inst()
```

We then define the function `run` which executes stateful computations.

```
 1 local run = function(init, task)
 2   local step = handler {
 3     val = function(_) return function() end end,
 4     [Get] = function(_, k)
 5       return function(s)
 6         return k(s)(s)
 7       end
 8     end,
 9     [Put] = function(s, k)
10       return function(_)
11         return k()(s)
12       end
13     end
14   }
15
16   return step(task)(init)
17 end
```

The function takes two arguments `init` for the initial state and a thunk `task` for the stateful computation. It first defines the handler `step`, which manipulates the normal-return case and the two effects labeled by `Get` and `Put`. Following the state-passing scheme, the value handler returns a function which ignores its argument (for state). In the stateful computation, when the effect `Get` is invoked, then the handler returns the function that retrieves the current state

`s` and supplies it to the current continuation (`k(s)` on Line 6) with the same state `s`. When the effect `Put` with an parameter `s` is invoked, the handler returns a thunk in which a meaningless value `()` is passed to the continuation, but a new state `s` is installed (Line 11). After defining the handler, the function `run` executes the computation `task` with the initial state `init` (Line 16).

It is important to note that the captured continuation is surrounded by the same handler `step`. In fact, the algebraic effects and handler are similar to the control operators shift0 and reset0 [19]; when an effect invoked by shift0 is captured by reset0, the captured delimited continuation is surrounded by the delimiter reset0.

2.3 Expressing Other Computational Effects

We can express other control effects using one-shot algebraic effects and handlers. Examples include generators and iterators, let-insertion in partial evaluation, and Go language's `defer`[7], We have already implemented by our library more advanced examples such as async/await, shift/reset, fetching the current time (a sort of dependency injection) and measuring the execution time. See our GitHub repository (see Footnote 2 and 3).

3 Embedding Algebraic Effects into Coroutines

This section explains our translation from one-shot algebraic effects and handlers to asymmetric coroutines. For this purpose, we define λ_{eff}, a language which has *one-shot* AEH, and λ_{ac}, a language which has asymmetric coroutines [20]. We then translate λ_{eff} to λ_{ac}, and show that it is a macro-expressible translation.

3.1 λ_{eff}

λ_{eff} is an untyped language with one-shot AEH based on Effy [23]. For simplicity, we omit dynamic creation of effect lablels.

Figure 1 defines the syntax of λ_{eff}. The set *Effects* is a finite set of effect lables, and we use *eff* as a meta variable for it. The syntactic categories v, e, and h, resp. represent values, expressions and handler expressions, resp. The expression `perform` *eff* v invokes the effect *eff* with the argument v, and `with` v `handle` e evaluates e under the handler. A `let` binding is written as `let` $x = c_1$ `in` c_2.

The handler expression `handler` *eff* (`val` $x \rightarrow e_1$) $((y, k) \rightarrow e_2)$ creates a handler which catches the effect *eff* and returns the value of e_2 where y is bound to the argument of the effect-performing operation, and k is bound to the delimited continuation when the effect is invoked. The expression `val` $x \rightarrow e_1$ gives a value handler, namely, a handler which is used when the body of a handler returns normally (no invocation of effects). For simplicity, λ_{eff} can handle only

[7] https://golang.org/ref/spec#Defer_statements.

$$
\begin{array}{rcl}
x & \in & \mathit{Variables} \\
\mathit{eff} & \in & \mathit{Effects} \\
v & ::= & x \mid h \mid \lambda x.\ e \\
e & ::= & v \mid v\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & & \mid \texttt{perform}\ \mathit{eff}\ \ v \mid \texttt{with}\ v\ \texttt{handle}\ e \\
h & ::= & \texttt{handler}\ \mathit{eff}\ \ (\texttt{val}\ \ x \rightarrow e)\ \ ((x, x) \rightarrow e) \\
 & & \\
w & ::= & \texttt{clos}\,(\lambda x.e, E) \mid \texttt{closh}\,(h, E) \\
F & ::= & (\Box\ e, E) \mid w\ \Box \\
 & & \mid (\texttt{let}\ x = \Box\ \texttt{in}\ e, E) \\
 & & \mid (\texttt{with}\ w\ \texttt{handle}\ \Box)^{\mathit{eff}} \\
 & & \mid (\texttt{with}\ \Box\ \texttt{handle}\ e, E) \\
C & ::= & e \mid w \\
E & ::= & [] \mid (x = w) :: E \\
K & ::= & [] \mid F :: K
\end{array}
$$

Fig. 1. Syntax and runtime representation of λ_{eff}

one effect per handler, whereas handlers in Effy can cope with multiple effects. But the latter can be simulated by our single-effect handlers, and our library actually provides the multi-effect variant; see Sect. 4.

The syntactic category w and the subsequent lines are used to define the semantics of λ_{eff}. The class w represents runtime values for function closures ($\texttt{clos}\,(\lambda x.e, E)$) and handlers ($\texttt{closh}\,(h, E)$) where E is a runtime environment, and F represents a *frame*, or a singular context, which means a 'one-step' fragment of a continuation. A (delimited) continuation K is a list of frames.

The call-by-value operational semantics of λ_{eff} is defined in the CEK-machine style [9]. Here we informally explain the semantics of effect primitives. Its details can be found in the full version of this paper; see our GitHub repository.

The handler expression $\texttt{handler}\ \mathit{eff}\ \ (\texttt{val}\ x \rightarrow e_v)\,((x, k) \rightarrow e_{\mathit{ef}}\ )$ creates a handler which consists of a value handler and an effect handler, and associates the effect label eff to it. The expression $\texttt{with}\ h\ \texttt{handle}\ e$ (called a handling expression) evaluates the expression e under the handler h. The expression $\texttt{perform}\ \mathit{eff}\ \ v$ invokes the effect eff with an argument v. Note that handling expressions may be nested, and an effect invocation is handled by the nearest (innermost) handler which can handle the effect. When the handled expression is evaluated to a value, the value handler is used.

3.2 λ_{ac}

The seminal work by de Moura and Ierusalimschy [20] classified various forms of coroutines found in programming languages, and formalized calculi for symmetric coroutines and asymmetric coroutines. The former represents classic coroutines which can call (resume) other coroutines, but coroutines cannot return to their callers. The latter represents modern coroutines where the caller-callee relation exists, hence, coroutines may return to their callers.

The language λ_{ac} is based on asymmetric coroutines[8]. For the purpose of translation and practical programming, we have added to this language several constructs such as data constructors, `let` with recursion, pattern matching, and comparison operators.

Figure 2 defines the syntax of λ_{ac}. The syntactic categories K and l, resp., represent data constructors and labels for coroutines, resp. The set *eff* corresponds to the set of effect labels in λ_{eff}, and we assume that its elements are constants in λ_{ac}. Values v are either constants, expressions formed by applying a data constructor to values $K\ \overrightarrow{v}^{\star}$, labels, variables, or lambda expressions. Expressions e are those in lambda calculus extended with pattern matching and mutual recursion, plus those for asymmetric coroutines: $l : e$ for a labeled expression which represents the "return point" of resuming a coroutine, `create` e for creating a coroutine and returning its label, `resume` $e_1\ e_2$ for resuming a coroutine, and `yield` e for yielding a value and returning to the caller of the current coroutine. $f\ \overrightarrow{x}$ is an abbreviation of $f\ x_1\ \cdots\ x_n$ and $\overrightarrow{\texttt{and}\ g\ \overrightarrow{y} = e}$ is of `and` $g_1\ \overrightarrow{y} = e_1\ \cdots$ `and` $g_n\ \overrightarrow{y} = e_n$. Similar abbreviation is used for constructors and pattern matching. The expression `match` e `with` *cases* is for pattern matching. We allow (restricted) guards in pattern matching so that *cases* $\overrightarrow{pat \to e}$ may have a guard `when` $x = x$.

The call-by-value operational semantics of λ_{ac} is defined in the same way as de Moura and Ierusalimschy, which is given in the full version of this paper. Here we briefly explain the semantics of the primitives for coroutines. `create` e creates a fresh label and a coroutine with its body being the value of e, and returns the label. The expression `resume` $l\ v$ resumes the coroutine labeled with l with the argument v. It is an error if a coroutine whose label is l does not exist, or has already been called. A resumed coroutine must return to the caller, so we create an expression $l : e'$ where e' is the body of the resumed coroutine. When an expression `yield` v is called during the evaluation of a coroutine, the coroutine is suspended and stored for future use, and v is returned to the caller of the current coroutine. It is an error if there is no caller of the current coroutine when `yield` is invoked.

3.3 Translation from λ_{eff} to λ_{ac}

This section presents a program translation from λ_{eff} to λ_{ac}, which is syntax-directed and compositional. The translation essentially does two things: to emu-

[8] Strictly speaking, our calculus is the one for *stackful* asymmetric coroutines according to de Moura and Ierusalimschy's classification.

$$
\begin{array}{rcl}
x & \in & \mathit{Variables} \\
K & \in & \{\mathit{Eff}, \mathit{Resend}, \mathit{True}, \mathit{False}\} \\
l & \in & \mathit{Labels} \\
\mathit{eff} & \in & \mathit{Effects} \\
v & ::= & \texttt{nil} \mid \mathit{eff} \mid K\ \overrightarrow{v} \mid l \mid x \mid \lambda x.e \\
e & ::= & v \mid K\ \overrightarrow{e} \mid l : e \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & & \mid \texttt{match}\ e\ \texttt{with}\ \mathit{cases} \\
 & & \mid \texttt{create}\ e \mid \texttt{resume}\ e\ e \mid \texttt{yield}\ e \\
\mathit{letrec} & ::= & \texttt{let rec}\ x\ \overrightarrow{x} = e\ \left[\overrightarrow{\texttt{and}\ x\ \overrightarrow{x} = e}\right]\ \texttt{in}\ e \\
\mathit{cases} & ::= & \overrightarrow{\mathit{pat}\ [\mathit{guard}] \to e} \\
\mathit{guard} & ::= & \texttt{when}\ x = x \\
\mathit{pat} & ::= & K\ \overrightarrow{\mathit{pat}} \mid x
\end{array}
$$

Fig. 2. Syntax of λ_{ac}

late multiple-effects AEH by a single-effect AEH, and then to emulate a single-effect AEH by an asymmetric coroutine. The first one is done by adding tags to distinguish different effects and by forwarding (resending) the raised effect if its tag does not match the tag of the handler. The second one is done by emulating a (one-shot) delimited continuation by an asymmetric coroutine. To save space, we give the whole translation as a single translation.

The whole translation is defined in Fig. 3 where a λ_{eff}-term e is translated to a λ_{ac}-term $[\![e]\!]$. The translation is homomorphic for a variable, a λ-abstraction, an application, and the `let` expression. An effect label *eff* is translated to a constant with the same name.

We translate `perform` to `yield` based on the following observation. In the calculus for AEH, when an effect is invoked, the control is transferred to a handler corresponding to the effect, while in the calculus for coroutines, when a `yield` is called, the control is transferred to its parent coroutine. Hence we can emulate the behavior of `perform` by `yield`. The translation wraps the arguments of `perform` with the tag *Eff*. This tag is used to determine whether the effect has been *yield*ed from the handled expression itself, or it has been resent by the handler. The handling expression `with` h `handle` e is translated to a simple application as the handler is translated to a function.

The translation for a handler (the last case in Fig. 3) is intricate, and we shall explain it by an example.

Consider the program M with the effects C_1, C_2, and C_3 in Fig. 4. Here we assume that our calculus is extended to have natural numbers arithmetic operations. Then M is translated to the program in Fig. 5 where some variables and `let`-bindings are renamed or inlined for readability.

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![\lambda x.e]\!] &= \lambda x.[\![e]\!] \\
[\![v_1\ v_2]\!] &= ([\![v_1]\!])\ ([\![v_2]\!]) \\
[\![\texttt{let}\ x = e\ \texttt{in}\ e']\!] &= \texttt{let}\ x = [\![e]\!]\ \texttt{in}\ [\![e']\!] \\
[\![\mathit{eff}]\!] &= \mathit{eff} \\
[\![\texttt{perform}\ \mathit{eff}\ v]\!] &= \texttt{yield}\ (\mathit{Eff}\ ([\![\mathit{eff}]\!])\ ([\![v]\!])) \\
[\![\texttt{with}\ h\ \texttt{handle}\ e]\!] &= [\![h]\!]\ (\lambda_.[\![e]\!]) \\
[\![\texttt{handler}\ \mathit{eff}\ (\texttt{val}\ x \to e_v)\ ((x,k) \to e_{\mathit{eff}})]\!] &= \\
&\quad \texttt{let}\ \mathit{eff} = [\![\mathit{eff}]\!]\ \texttt{in} \\
&\quad \texttt{let}\ \mathit{vh} = \lambda x.[\![e_v]\!]\ \texttt{in} \\
&\quad \texttt{let}\ \mathit{effh} = \lambda x\ k.[\![e_{\mathit{eff}}]\!]\ \texttt{in} \\
&\quad \mathit{handler}\ \mathit{eff}\ \mathit{vh}\ \mathit{effh}
\end{aligned}
$$

where $\mathit{handler} =$

$$
\begin{array}{l}
\texttt{let rec}\ \mathit{handler}\ \mathit{eff}\ \mathit{vh}\ \mathit{effh}\ \mathit{th} = \\
\quad \texttt{let}\ \mathit{co} = \texttt{create}\ \mathit{th}\ \texttt{in} \\
\quad \texttt{let rec}\ \mathit{continue}\ \mathit{arg} = \mathit{handle}\ (\texttt{resume}\ \mathit{co}\ \mathit{arg}) \\
\quad \texttt{and}\ \mathit{rehandle}\ k\ \mathit{arg} = \mathit{handler}\ \mathit{eff}\ \mathit{continue}\ \mathit{effh}\ (\lambda_.k\ \mathit{arg}) \\
\quad \texttt{and}\ \mathit{handle}\ r = \\
\qquad \texttt{match}\ r\ \texttt{with} \\
\qquad \begin{array}{lll}
\mid \mathit{Eff}\ \mathit{eff}'\ v & \texttt{when}\ \mathit{eff}' = \mathit{eff} & \to \mathit{effh}\ v\ \mathit{continue} \\
\mid \mathit{Eff}\ _\ _ & & \to \texttt{yield}\ (\mathit{Resend}\ r\ \mathit{continue}) \\
\mid \mathit{Resend}\ (\mathit{Eff}\ \mathit{eff}'\ v)\ k & \texttt{when}\ \mathit{eff}' = \mathit{eff} & \to \mathit{effh}\ v\ (\mathit{rehandle}\ k) \\
\mid \mathit{Resend}\ \mathit{effv}\ k & & \to \texttt{yield}\ (\mathit{Resend}\ \mathit{effv}\ (\mathit{rehandle}\ k)) \\
\mid _ & & \to \mathit{vh}\ r
\end{array} \\
\quad \texttt{in}\ \mathit{continue}\ \texttt{nil} \\
\texttt{in}\ \mathit{handler}
\end{array}
$$

Fig. 3. Translation from λ_{eff} to λ_{ac}

The term after translation $[\![M]\!]$ contains the function *handler* defined in Fig. 3, which works as follows: `handler` makes a coroutine from a thunk, defines three functions *continue*, *rehandle* and *handle*, and then evaluates *continue* `nil`. *continue* passes *arg* to *co*, `resumes` it, and passes the return value to *handle*. *handle* dispatches the process by the return value of `resume` according to the equivalence of tags and effect labels.

Let us evaluate $[\![M]\!]$. It first binds h_1, h_2, and h_3 to the values of applying *handler* to three arguments. Since the function *handler* needs four arguments (see Fig. 3), the values of h_i are still closures. After setting h_i, we evaluate $h_3(\lambda_. \cdots)$, which triggers the actual computation of the body of *handler*. It then creates a new coroutine for the argument of h_3, defines several functions and

```
M = let h1 = handler C1
      (val v → v) ((x, k) → k x) in
    let h2 = handler C2
      (val v → v) ((x, k) → k x) in
    let h3 = handler C3
      (val v → v) ((x, k) → k x) in
    with h3 handle
    with h2 handle
    with h1 handle
      let a = perform (C1 10) in
      let b = perform (C3 17) in
      a + b
```

Fig. 4. Example program in λ_{eff}

```
⟦M⟧ = let h1 = let vh1 = λv. v in
                 let effh1 = λx. λk. k v in
                 handler C1 vh1 effh1 in
      let h2 = let vh2 = λv. v in
                 let effh2 = λx. λk. k v in
                 handler C2 vh2 effh2 in
      let h3 = let vh3 = λv. v in
                 let effh3 = λx. λk. k v in
                 handler C3 vh3 effh3 in
      h3 (λ_. h2 (λ_. h1 (λ_.
        let a = yield (Eff C1 10) in
        let b = yield (Eff C3 17) in
        a + b )))
```

Fig. 5. Example after translation

evaluates *continue nil*, which in effect applies the argument of h_3 (the thunk $\lambda_.\cdots$) to *nil*. Similarly, $h_2(\lambda_.\cdots)$ and then $h_1(\lambda_.\cdots)$ are evaluated, and finally the subterm `yield` (*Eff* C_1 10) is evaluated, and a value is yielded. It is caught by the innermost handler h_1. Since it has the tag *Eff* and h_1 can handle C_1, the first out of five patterns in *handle* matches it, and *effh* 10 *continue* is evaluated. By passing *continue* as the continuation, the computation of a handled expression can be resumed, which is suspended at the yielded position. Since *continue* passes the return value of `resume` to *handle*, the effect can be handled by the same handler again. Hence a is bound to 10.

The above case uses only the first pattern in the function *handle* which corresponds to a single effect. The handler needs to treat more involved cases when AEH allows multiple effects, which we shall explain shortly.

Continuing the evaluation of $[\![M]\!]$, we encounter the term `yield` (*Eff* C_3 17). When it is executed, the handler h_1 catches it, but h_1 cannot handle C_3. Hence the second pattern of *handle* matches, and another *yield* term with the tag *Resend* is evaluated which is caught by the next outside handler, namely, h_2. Since it has the tag *Resend* and h_2 cannot handle C_3, the fourth pattern of *handle* matches, which again yields the value with the tag *Resend*. A difference from the previous case is that the function *rehandle* is applied to k, where *rehandle* creates a handler that handles the thunk of the application of two given arguments. By setting *continue* to the value handler, the computation of the current handling expression can be resumed when the computation of the *rehandle* passed as a continuation is finished. *rehandle* has another role which adjusts the layers of the coroutines. In the second clause of *handle*, *yield* is called, so the execution exits from the coroutine.

The resent effect is captured by h_3 again. Now it matches the third pattern of *handle*, and similarly to the above case, *rehandle* k is passed to *effh* as a continuation. Then it returns 17, to which b is bound, and the value of the handled expression is 27. Then h_1 receives it, and the fifth wildcard pattern of *handle* matches, and the value of the entire expression is 27.

Although our translation looks complicated, we emphasize that our translation is compositional and local, syntax-directed, and needs only basic functionality of asymmetric coroutines. We also note that it does not rely on higher-order stores or other fancy features unlike de Moura and Ierusalimschy's work.[9] With this simplicity, several programmers have already ported our translation to other languages than Ruby and Lua.

3.4 Macro-expressible Translation

We will claim that the translation from λ_{eff} to λ_{ac} in the previous section is simple and efficient. To support the former claim, this subsection shows that it is a macro-expressible translation in the sense of Felleisen. The latter claim will be discussed in the subsequent section.

Felleisen studied the notion of macro expressivity, which is a more fine-grained notion than most others to measure the expressive power of language primitives [8]. For instance, *call*/*cc* (call-with-current-continuation) can be translated away by a CPS translation to a pure lambda calculus, yet, it is not macro-expressible in pure lambda calculus since the translation is global and not macro expressible. On the other hand, a simple let expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ can be locally translated by $(\lambda x.e_2)\ e_1$, therefore, it is macro-expressible in the pure lambda calculus.

While Felleisen defined the notion for the setting where a language L_1 is a proper extension of another language L_2, we want to compare the expressive power of two languages L_1 and L_2 where L_1 and L_2 are extensions of a common language L_0. To deal with this setting, we use Forster et al.'s definition for the macro-expressible translation [10], and we give its slightly simplified version here.

Definition 1 (Macro-expressible translation). *Let L_0 be a language, and L_1 and L_2, resp., be the language L_0 augmented with a set of primitives $X_1, \cdots, X_n$ and $Y_1, \cdots, Y_m$, resp. A translation ϕ from L_1 to L_2 is a macro-expressible translation if and only if all of the following conditions hold.*

- *ϕ is homomorphic for the primitives in L_0. For instance, if a binary infix operator $\oplus$ is in L_0, then $\phi\,(e_1 \oplus e_2)$ is $\phi\,(e_1) \oplus \phi\,(e_2)$.*
- *ϕ maps each X_i of arity n to a syntactic expression M_i in L_2 which has n free variables $x_1, \cdots, x_n$ such that the following holds:*

$$\phi\,(X_i(e_1, \cdots, e_n)) = M_i\,[\phi\,(e_1)\,/x_1, \cdots, \phi\,(e_n)\,/x_n]$$

The expression in the right-hand side represents simultaneous substitution for the variables $x_1, \cdots, x_n$ in M_i.

[9] See Sect. 5.3 in [20].

To state the above definition we have made two simplifications. First, the equality in this definition should be, in general, semantic equality where we assume that each language is equipped with a certain semantics, but in this paper, we can regard it as syntactic equality. Second, we do not consider the case when X_i works as a binder such as the `let` expression[10], but we do not need to consider such cases.

It is easy to show that our translation in the previous subsection conforms the conditions for a macro-expressible translation.

Theorem 1. *Our translation in Fig. 3 is a macro-expressible translation.*

Proof Sketch. It is easy to check that our translation $[\![\cdot]\!]$ is homomorphic for the variable, lambda abstraction, application, let, the effect expression.

For the primitives of algebraic effects and handlers, we need to check each case. For the primitive `perform`, let M be `yield` $(\mathit{Eff}\ x_1\ x_2)$, then we have $[\![\texttt{perform}\ e_1\ \ e_2]\!] = M\,[[\![e_1]\!]/x_1,\ [\![e_2]\!]/x_2]$, and we are done. Other cases are similar. □

As we wrote above, a macro-expressible translation is rather discriminating, or sensitive to small differences between language primitives. Only local translations are macro-expressible translation. Since global translations such as a CPS translation and a state-passing translation do not qualify as macro-expressible, state and first-class continuations are not macro-expressible in pure lambda calculus.

Put differently, if we have a macro-expressible translation for a primitive X in a language L_0, then we can implement X using the translation without changing any other primitives in L_0 This is a simple, but rather important property for our work, as it is a necessary condition to implement X as a simple library in L_0, unless we have an access to language's run-time, or reification is allowed.

4 Implementation

We have implemented AEH in Lua and Ruby based on the translation in Sect. 3. Since the translation is macro-expressible, we can realize our implementation as a simple library. Our implementations are compact. The Lua library is implemented in 160 lines and the core of the Ruby library is in 340 lines, even including comments for documentation generation. All our code is available via GitHub.

Several issues have arisen in the process of implementation which we will explain below.

Multiple Effect Handlers: Our calculus λ_{eff} has the restriction that a handler can catch only one effect. However, this restriction is only for the presentation purpose, and in our actual implementation, one handler may catch multiple effects, All examples including the examples in this paper that use multiple effects per handler run without problems using our library. We also note that there is no critical performance downgrade of having multiple effects per handler.

[10] Felleisen considers the case where each argument may be bound by the construct.

Dynamic Effect Creation: In the language λ_{eff}, we have no way to create new effect labels dynamically. Again this is due to simplicity, and we have eliminated this restriction in our implementation. The merit of allowing dynamic creation of effect instances is that a certain kind of effectful programs require effect instances to be mutually distinct, for instance, higher-order effects [16].

Conflict with Other Effects: An assumption on our translation is that all effects are written via AEH. If our source program uses other effects than AEH, it will cause a problem as they may interfere with the internally used coroutines. For instance, if we use our library in Lua, and simultaneously use Lua's native coroutine directly, yielding a value in the source program may be accidentally caught by an internal coroutine. As consequence, we must not use native coroutines with our library for AEH.

However, this problem can be solved in the following way, thanks to the expressivity of AEH. See the following code.

```
1  local Yield = inst()
2
3  local yield = function(v)
4    return perform(Yield, v)
5  end
6
7  local create = function(f)
8    return { it = f, handled = false }
9  end
10
11 local resume = function(co, v)
12   if co.handled then
13     return co.it(v)
14   else
15     co.handled = true
16     return handler({
17       val = function(x) return x end,
18       [Yield] = function(u, k)
19         co.it = k
20         return u
21       end
22     })(function() return co.it(v) end)
23   end
24 end
```

The code implements asymmetric coroutines by algebraic effects and handlers in Lua. The function `yield` should throw a value to `resume`, so `yield` should be an effect invocation and `resume` should be a handler. This correspondence is the inverse of the translation in Fig. 3. So we define the effect `Yield` (Line 1) and the function `yield` (Line 3) as a wrapper for the invocation of the effect.

The function `create` (Line 7) creates a reference cell by a table. We represent a coroutine as a reference cell, which is initialized to the function `f` and the flag `handled` explained later. The handler `resume` (Line 11) catches the effect `Yield` with an argument and a continuation. This continuation is the rest of computation of the coroutine, so the handler stores the continuation to the cell and returns the value `u` (Lines 19 and 20). Since we provide a deep handler, it is not necessary to set the handler multiple times. The flag `handled` asserts if the function is handled by the handler (Line 12). The function `resume` checks the flag; if the flag is not set, it turns on the flag and runs the function with the handler. Otherwise, it runs the function alone.

Although we believe that the above technique may be used for other computational effects, it is left for future work to combine them with algebraic effect and handlers to obtain an efficient implementation.

5 Evaluation

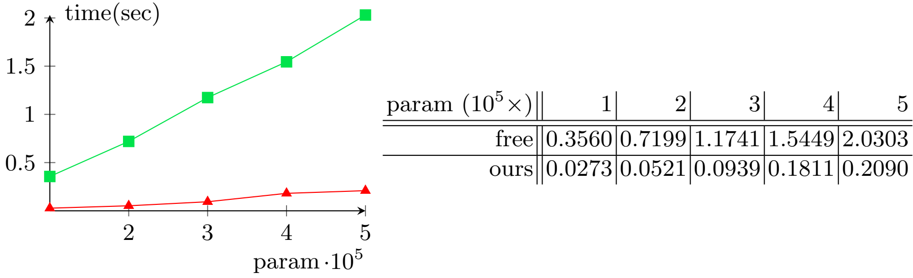
We have conducted experiments on microbenchmark using our library in Lua, and implementation in Lua based on free monads [23], and compared their performance. All the code for the benchmark is publicly available in the GitHub repository[11]. In the following figures, the symbol ▲ represents the result of our library, and ■ the free-monad based implementation. One of the benchmarks compares to native coroutines of Lua and indicates the result as the symbol ⋆ in a graph. The experiments have been conducted on the environment in Table 1.

Table 1. Environment for benchmark

OS	Arch Linux
CPU	Intel Core i7-8565U
Main memory	16 GB DDR4
Lua processor	LuaJIT 2.05

Figure 6 is the result of the benchmark for emulating a state monad. The benchmark uses the function `count`, cited from [14], adjusted for our library and free monad, which recursively runs a simple computation consisting of single-layer, single-effect handlers for the number of times as the input parameter. The result shows that our library is approx. 10 times faster than the free-monad based implementation for this simple case. The reason why free monads are rather slow is that the bind operator requires a continuation as the next action, but the cost for creating function closures is rather high for imperative languages such as Lua. Also, functional languages such as Haskell may offer optimization for free monads, while the benchmark uses naive implementation. Nevertheless, the results are encouraging for our embedding.

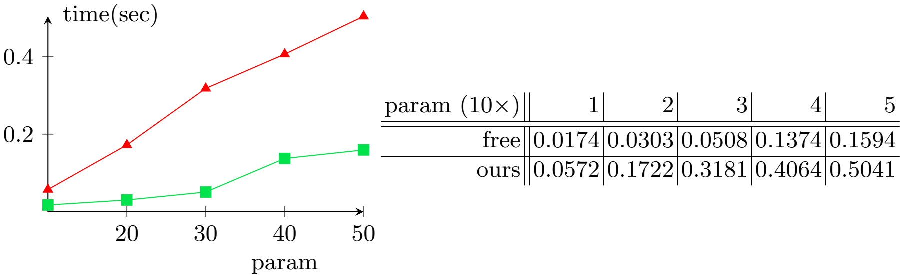
[11] https://github.com/nymphium/effs-benchmark.



param ($10^5\times$)	1	2	3	4	5
free	0.3560	0.7199	1.1741	1.5449	2.0303
ours	0.0273	0.0521	0.0939	0.1811	0.2090

Fig. 6. Result of `onestate` benchmark

In the experiments in Fig. 7, the benchmark program runs the function `count` 3,000 times in deeply nested handlers. The parameter in the table corresponds to the number of nested handlers/coroutines, hence 50 (the right-most column) is an extreme case. As expected, our library runs three times slower than the free monad does for this case. The reason is that *rehandle* creates a new coroutine, which is called every time an effect is caught from the other handler shown in Fig. 3, so it degrades the performance.

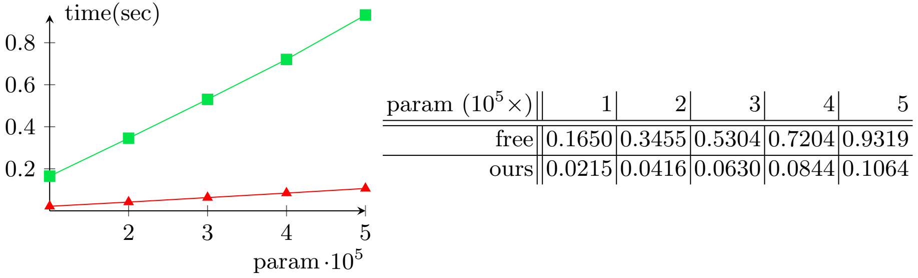


param ($10\times$)	1	2	3	4	5
free	0.0174	0.0303	0.0508	0.1374	0.1594
ours	0.0572	0.1722	0.3181	0.4064	0.5041

Fig. 7. Result of `multistate` benchmark

The next experiment executed a `for`-loop, where the number of iteration is given as a parameter in the table of Fig. 8. The benchmark program sets a handler outside of the loop, and invokes an effect in the `for`-loop. Our library runs 9 times as fast as the free-monad based implementation. Note that free monads need the `forM`-operator which has large overhead. Again an advanced compiler may reduce the overhead.

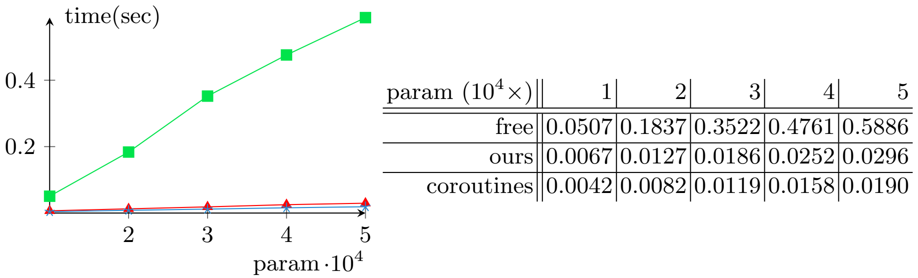
Figure 9 shows the result of the benchmark, which solves the same-fringe problem [11] by using algebraic effects and coroutines. The problem is to determine whether given two trees have the same “fringe”, an enumeration of leaves of the tree in a certain order. The benchmark is given the number of leaves as a parameter. We implemented it by algebraic effects and handlers via free monad, and via our library, and the result is shown in Sect. 4. We also implemented the



param ($10^5\times$)	1	2	3	4	5
free	0.1650	0.3455	0.5304	0.7204	0.9319
ours	0.0215	0.0416	0.0630	0.0844	0.1064

Fig. 8. Result of `looper` benchmark

solver with native coroutines of Lua. Our library yields 18 times performance gain compared to the free-monad method. Remarkably, our library is only 1.6 times slower than native coroutines, despite the overhead caused by the double translations.



param ($10^4\times$)	1	2	3	4	5
free	0.0507	0.1837	0.3522	0.4761	0.5886
ours	0.0067	0.0127	0.0186	0.0252	0.0296
coroutines	0.0042	0.0082	0.0119	0.0158	0.0190

Fig. 9. Result of `same_fringe` benchmark

In summary, our way of implementing AEH is advantageous in several programming languages from the performance viewpoint. We also emphasize that writing effectful programs using coroutines is harder than writing the same programs using AEH, which provide high-level abstraction.

6 Related Work and Discussion

In this section, we discuss closely related work which has not been mentioned in this paper and picks up a few important issues for discussion.

Shallow Handler: We have shown the embedding with *deep handlers*, which inserts the handler to the topmost position of captured delimited continuations. In the literature, there has been discussion on the merits and demerits between

deep handlers and *shallow handlers* [12], which do not insert the handler to captured delimited continuations, hence an effect invocation during the execution of a delimited continuation is not captured by the same handler. We have also implemented the shallow handler with coroutines shown in Fig. 10. The idea is simple; after a handler catches an effect, it resends any effects to an outer handler. We have already explained the role of *rehandle* in Fig. 3, namely, it adjusts the layer of coroutines, and handles the effect invocation in the continuation. For the shallow handlers, the former is necessary but the latter is not, hence we have prepared simplified functions $continue_0$ and $rehandle_0$.

$$
\begin{array}{l}
[\![\texttt{handler}^\dagger\ \mathit{eff}\ \ (\texttt{val}\ x \to e_v)\ \ ((x,k) \to e_{\mathit{eff}})]\!] = \\
\qquad\qquad\qquad\qquad \texttt{let}\ \mathit{eff} = [\![\mathit{eff}]\!]\ \texttt{in} \\
\qquad\qquad\qquad\qquad \texttt{let}\ \mathit{vh} = \lambda x.[\![e_v]\!]\ \texttt{in} \\
\qquad\qquad\qquad\qquad \texttt{let}\ \mathit{effh} = \lambda x\ k.[\![e_{\mathit{eff}}]\!]\ \texttt{in} \\
\qquad\qquad\qquad\qquad \mathit{handler}^\dagger\ \mathit{eff}\ \mathit{vh}\ \mathit{effh} \\
\text{where } \mathit{handler}^\dagger = \\
\quad \texttt{let rec}\ \mathit{handler}\ \mathit{eff}\ \mathit{vh}\ \mathit{effh}\ \mathit{th} = \\
\quad\quad \texttt{let}\ \mathit{co} = \texttt{create}\ \mathit{th}\ \texttt{in} \\
\quad\quad \texttt{let rec}\ \mathit{continue}\ \mathit{arg} = \mathit{handle}\ \ (\texttt{resume}\ \mathit{co}\ \mathit{arg}) \\
\quad\quad \texttt{and}\ \mathit{rehandle}\ k\ \mathit{arg} = \mathit{handler}\ \mathit{eff}\ \mathit{continue}\ \mathit{effh}\ \ (\lambda_.k\ \mathit{arg}) \\
\quad\quad \texttt{and}\ \mathit{continue}_0 = \texttt{resume}\ \mathit{co} \\
\quad\quad \texttt{and}\ \mathit{rehandle}_0\ k = \texttt{resume}\ \ (\texttt{create}\ k) \\
\quad\quad \texttt{and}\ \mathit{handle}\ r = \\
\quad\quad\quad \texttt{match}\ r\ \texttt{with} \\
\quad\quad\quad \begin{array}{lll}
\mid \mathit{Eff}\ \ \mathit{eff}'\ v & \texttt{when}\ \mathit{eff}' = \mathit{eff} & \to \mathit{effh}\ v\ \mathit{continue}_0 \\
\mid \mathit{Eff}\ _\ _ & & \to \texttt{yield}\ \ (\mathit{Resend}\ r\ \mathit{continue}) \\
\mid \mathit{Resend}\ \ (\mathit{Eff}\ \ \mathit{eff}'\ v)\ \ k & \texttt{when}\ \mathit{eff}' = \mathit{eff} & \to \mathit{effh}\ v\ \ (\mathit{rehandle}\ k) \\
\mid \mathit{Resend}\ \ \mathit{effv}\ k & & \to \texttt{yield}\ \ (\mathit{Resend}\ \ \mathit{effv}\ \ (\mathit{rehandle}_0\ k)) \\
\mid _ & & \to \mathit{vh}\ r
\end{array} \\
\quad\quad \texttt{in}\ \mathit{continue}\ \texttt{nil} \\
\quad \texttt{in}\ \mathit{handler}
\end{array}
$$

Fig. 10. Translation from shallow handlers to coroutines

One-Shot Continuations: We are not the first to study the one-shot variant of control operators. Bruggeman et al. gave a one-shot control operator *call/1cc* under the observation that most continuations are run at most once [5]. They showed that, by replacing *call/cc* by *call/1cc*, programs can be executed with less memory consumption and higher performance. Berdine et al. introduced

a linear type system as static approximation of one-shotness, and showed that many control abstractions may be typed by their type system [2].

James and Sabry studied the *yield* operator for generators, and proved that it is as expressive as one-shot delimited continuations [13]. They also introduced a generalized *yield* operator for multi-shot continuations and showed the connection between it and delimited-control operators.

Multicore OCaml is a dialect of OCaml which natively supports algebraic effects by runtime stack manipulation. Its motivation is to write concurrent programming in direct style [7]. It provides one-shot continuations, and if multi-shot continuations are needed, they allow explicit copying of continuations.

Free Monad: We have already compared our work with free-monad based implementations of algebraic effects and handlers. On the positive side, it gives a systematic and elegant implementation for various effects. Its downside is significant overhead in performance. Note that our embedding-based implementation does not interfere with surface languages, while free-monad based implementations force a programmer to use the monadic style. With our implementation, the surface language with algebraic effects and handlers can be presented in direct style and monadic style.

7 Conclusion

We have presented a novel embedding technique for algebraic effects and handlers into asymmetric coroutines, and shown a translation from the former to the latter as a simple, direct, syntax-directed compositional translation. Compared with other implementation methods, our technique is applicable to many languages which have asymmetric coroutines. We have demonstrated the applicability of our embedding by implementing the libraries in Lua and Ruby. Our technique seems to be attractive for other researchers, and some of them have implemented our translation for other languages such as JavaScript and Rust. We expect that the simplicity of our implementation is advantageous to be used by more people, more languages, and more applications.

The key of our development is the one-shotness restriction of continuations. Our embedding relies on the fact that, many applications with computational effects use continuations at most once, and they allow more efficient implementation than the general, multi-shot continuations. One-shotness is a dynamic property, and its static approximation, linearly used (delimited) continuations, or linear continuation-passing style, are studied in the literature. We hope that the formal foundation of this paper's result is studied more deeply, and coroutines and their connection with other control operators find a solid theoretical foundation.

We briefly mention future work. There are many directions to extend our work. Of particular interest is to prove the semantics preservation of our translation. Introducing an appropriate type system is also an interesting next step. Another exciting issue is to relate and compare various control abstractions in

the literature and in the practical programming languages. For instance, React, a popular web framework for JavaScript, has a utility software Hooks[12], which allows programmers to build components with side-effects modularly. Abramov pointed out the relevance between Hooks and algebraic effects in his blog post[13], and we think that investigating this relationship based on our work is promising. Finally embedding algebraic effects and handlers in modern languages such as Rust (via generators) may lead to a composable, efficient and safe implementation of controlful programs.

Acknowledgement. We are grateful for the reviewers of earlier versions of this paper for constructive comments and numerous suggestions. The second author is supported in part by JSPS Grants-in-Aid (B) 18H03218.

References

1. Bauer, A., Pretnar, M.: Programming with Algebraic Effects and Handlers. J. Log. Algebr. Methods Program. **84**, 108–123 (2012)
2. Berdine, J., O'Hearn, P., Reddy, U., Thielecke, H.: Linear continuation-passing. Higher-Order Symbolic Comput. **15**, 181–208 (2002). https://doi.org/10.1023/A:1020891112409
3. Brachthäuser, J., Schuster, P.: Effekt: extensible algebraic effects in Scala (short paper). In: Proceedings of the 8th ACM SIGPLAN Symposium on Scala, pp. 67–72 (2017)
4. Brachthäuser, J., Schuster, P., Ostermann, K.: Effect handlers for the masses. Proc. ACM Program. Lang. **2**, 1–27 (2018)
5. Bruggeman, C., Waddell, O., Dybvig, R.: Representing control in the presence of one-shot continuations, vol. 31, pp. 99–107 (1970)
6. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, pp. 151–160 (1990)
7. Dolan, S., White, L., Madhavapeddy, A.: Multicore OCaml. In: OCaml Users and Developers Workshop (2014)
8. Felleisen, M.: On the Expressive Power of Programming Languages. In: Selected Papers from the Symposium on 3rd European Symposium on Programming, ESOP 1990, pp. 35–75. Elsevier North-Holland Inc., USA (1991)
9. Felleisen, M., Friedman, D.P.: Control operators, the SECD-machine, and the λ-calculus. In: Formal Description of Programming Concepts (1987)
10. Forster, Y., Kammar, O., Lindley, S., Pretnar, M.: On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. J. Funct. Program. **29**, e15 (2019). https://doi.org/10.1017/S0956796819000121
11. Gabriel, R.P.: The Design of Parallel Programming Languages, pp. 91–108. Academic Press Professional Inc., Cambridge (1991)
12. Hillerström, D., Lindley, S.: Shallow effect handlers. In: Ryu, S. (ed.) APLAS 2018. LNCS, vol. 11275, pp. 415–435. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02768-1_22

[12] https://reactjs.org/docs/hooks-reference.html.
[13] https://overreacted.io/algebraic-effects-for-the-rest-of-us/.

13. James, R., Sabry, A.: Yield: mainstream delimited continuations. In: Proceedings of Workshop on the Theory and Practice of Delimited Continuations, pp. 1–12 (2011)
14. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: ACM SIGPLAN International Conference on Functional Programming, pp. 145–158 (2013)
15. Kiselyov, O., Ishii, H.: Freer monads, more extensible effects. In: ACM SIGPLAN International Symposium on Haskell, pp. 94–105 (2015)
16. Kiselyov, O., Sivaramakrishnan, K.: Eff directly in OCaml. Electron. Proc. Theor. Comput. Sci. **285**, 23–58 (2018)
17. Leijen, D.: Algebraic effects for functional programming. Technical report, p. 15 (2016)
18. Leijen, D.: Implementing algebraic effects in C. In: Chang, B.-Y.E. (ed.) APLAS 2017. LNCS, vol. 10695, pp. 339–363. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71237-6_17
19. Materzok, M., Biernacki, D.: Subtyping delimited continuations. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, pp. 81–93. ACM (2011). https://doi.org/10.1145/2034773.2034786
20. Moura, A.D., Ierusalimschy, R.: Revisiting coroutines. ACM Trans. Program. Lang. Syst. **31**, 1–31 (2004)
21. Plotkin, G., Power, J.: Algebraic operations and generic effects. Appl. Categor. Struct. **11**, 69–94 (2003). https://doi.org/10.1023/A:1023064908962
22. Plotkin, G., Pretnar, M.: Handling algebraic effects. Log. Methods Comput. Sci. **9**, 1–36 (2013)
23. Pretnar, M., Saleh, A.H., Faes, A., Schrijvers, T.: Efficient compilation of algebraic effects and handlers. CW reports, volume CW708, Department of Computer Science, KU Leuven (2017)