One-shot Algebraic Effects as Coroutines[⋆]

Satoru Kawahara[1,2⋆⋆] and Yukiyoshi Kameyama[1,3]

[1] Department of Computer Science, University of Tsukuba, Japan
[2] `sat@logic.cs.tsukuba.ac.jp`
[3] `kameyama@acm.org`

Abstract. [Kwhr: fix 32] This paper presents a translation from algebraic effects and handlers to asymmetric coroutines, which provides simple, portable and widely applicable implementation of algebraic effects. Algebraic effects and handlers are an emerging new feature 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 it 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 Effects and Handlers · Coroutines · Continuations · Control Operators · Macro Expressibility

1 Introduction

Algebraic effects [21] and handlers [22] (AEH for short) are an emerging new feature to model effectful computations in a modular way. They are gaining more and more attention not only from researchers but also from practitioners. [Kwhr: fix 1] 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 implementation method for AEH which is available in many existing programming languages.

⋆ Research Paper

⋆⋆ Student

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; [Kwhr: fix 5 and 34] The implementation method based on stack manipulation is used for JVM bytecode and C [4,18], however, [Kwhr: fix 6 and 7] an implementer needs deep insight on its internal structure. It then follows that the implementation cost is rather high, which prevents the feature from being implemented in various language systems. [Kwhr: fix 5 and 34] 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, its major demerits are that it enforces a programmer to use monadic-style, and that it is often inefficient.

This paper presents a new systematic method of implementing algebraic effects and handlers which is general, available in many languages, and simple and portable, compared to the existing implementations. 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, 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, [Kwhr: fix 2] 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, [Kwhr: fix 31] thanks to the fact that no copying of continuations is necessary [5]. [Kwhr: fix I rewrote the following paragraph completely.] 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 with 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. Thanks to this property, we can implement AEH as a simple library, and we have created AEH libraries for Lua [4] and Ruby[5], which have been published via github. Our libraries have been used by several users, and interested users have ported our libraries to other languages[6][7].

Our main contributions in this paper are the following.

- [Kwhr: fix 12] We show an embedding of one-shot algebraic effects and handlers. We use standard asymmetric coroutines only, and no special control features

[4] https://github.com/nymphium/eff.lua
[5] https://github.com/nymphium/ruff
[6] https://github.com/MakeNowJust/eff.js
[7] https://github.com/pandaman64/effective-rust [Kwhr: fix 25] Since Rust does not have coroutines, `Future` is used instead.

are needed. Hence our embedding is applicable to various languages as long as they have asymmetric coroutines.
- [Kwhr: fix 13] 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.
- Our embedding is defined as 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. [Kwhr: fix 33] Algebraic effects and handlers is a complex system, and the implementation can be therefore error-prone. So our formalized implementation 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 [Kwhr: fix 39] 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 Section 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.

```lua
local DivideByZero = inst()
```

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

```lua
local div = function(x, y)
  if y == 0 then
    return perform(DivideByZero, nil)
  else
    return x / y
  end
end
```

This code snippet is Lua's definition for the function `div`, which takes two argument `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 [Kwhr: fix 30] `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 provides the function `handler` to create 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[8], as its sole argument in which `e1 = e2` represents the key-value pair `["e1"] = e2`. The first key-value pair (Line 2) has the key `val`, and defines a value handler which is used when no effect happens, and 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 (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
```

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

```
end)
```

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`, which is bound to the variable `k`. (Strictly speaking, the delimited continuation should be surrounded by the handler [Kwhr: fix 30] `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.

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

We then define the function `run` to execute stateful computations.

```
local run = function(init, task)
  local step = handler {
    val = function(_) return function() end end,
    [Get] = function(_, k)
      return function(s)
        return k(s)(s)
      end
    end,
    [Put] = function(s, k)
      return function(_)
        return k()(s)
      end
    end
  }

  return step(task)(init)
end
```

The function takes two arguments `init` for the initial state (such as a single value or a tuple of several values) 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, [Kwhr: fix 43] 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)` in 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).

Note that it is important 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 is invoked by shift0 and captured by reset0, the captured delimited continuation is surrounded by the delimiter reset0.

2.3 Expressing other Computational Effects

We can express other advanced control effects using one-shot algebraic effects and handlers. Examples include generators and iterators, let-insertion in partial evaluation, and Go language's `defer`[9], Due to lack of space, we cannot show these examples in this paper. See the github repository of our library. We have already implemented async/await, shift/reset, fetching current time (a sort of dependency injection) and measuring execution time, by our library.

3 Embedding Algebraic Effects with 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. 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 meta variables 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 specified by the value v. A usual `let` binding is written as `let` $x = c_1$ `in` c_2.

The handler expression `handler` *eff* $(\texttt{val}\ x \rightarrow e_1)\,((y, k) \rightarrow e_2)$ 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

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

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

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

handler, namely, a handler which is used when the body of a handler returns normally (does not invoke an effect). For simplicity, λ_{eff} can handle only 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 Section 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 enviroment, and [Kwhr: fix 47] 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.

[Kwhr: fix 45] The call-by-value operational semantics of λ_{eff} is defined in the CEK-machine style [9]. We give it in Section A of the appendix of this paper, and here we informally explain the effect primitives only. The handler expression $\texttt{handler}\ \mathit{eff}\ \ (\texttt{val}\ x \to e_v)\,((x, k) \to e_{ef}\ )$ creates a handler which consists of a value handler and an effect handler, and associates the effect label eff to it. The expression $\texttt{with}\ h\ \texttt{handle}\ e$ (which is 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 caught (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}

De Moura and Ierusalimschy's seminal work [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-caller relation exists, hence, coroutines may return to their callers.

The language λ_{ac} is based on asymmetric coroutines[10]. 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, an expression 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 conditional expressions, 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 (calling) 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 $f\ x_0\ x_1\ \cdots\cdots\ x_n$ and $\overrightarrow{\texttt{and}\ g\ \overrightarrow{y} = e}$ is of `and` $g_0\ \overrightarrow{y} = e_0$ `and` $g_1\ \overrightarrow{y} = e_1$ `and` $\cdots\cdots$ `and` $g_m\ \overrightarrow{y} = e_m$. A similar abbreviation is applied to constructors and pattern matching.

The expression `match` e `with` *cases* is for pattern matching. We add restricted guards to pattern matching so that *cases* may contain a form $K\ \overrightarrow{x}$ `when` $x = x \to e$. This restricted form is sufficient for our purpose.

The call-by-value operational semantics of λ_{ac} is defined in the same way as de Moura and Ierusalimschy and given in Section B of Appendix. Here we briefly explain the semantics of the primitives for coroutine; `create` e creates a unique 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 against 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 $l : e_3$ where e_e is the body of the resumed coroutine. When an expression `yield` v is called in 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.

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

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

Fig. 2: the syntax of λ_{ac}

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

We present a program translation from λ_{eff} to λ_{ac}, which is syntax-directed and [Kwhr: fix 30] compositional. The whole translation is defined in Figure 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 behaviour of `perform` by `yield`. The translation wraps the arguments of `perform` with the tag *Eff* and translates them. [Kwhr: fix 50] This tag is used to determine whether the effect has been *yield*ed from the handled expression itself, or the effect has been resent (forwarded) by the handler. The handling expression `with` h `handle` e is translated to a simple application as the handler is mapped to a function.

The translation for a handler (the last case in Figure 3) is highly non trivial, and we shall explain it using an example.

Consider the program M in λ_{eff} with the effects C_1, C_2, and C_3 (Figure 4). Here we assume that our calculus is extended to have natural numbers arithmetic operations. Then M is translated to the program in Figure 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 $handler$ =
  let rec $handler\ eff\ vh\ effh\ th$ =
    let $co$ = create $th$ in
    let rec $continue\ arg$ = $handle$ (resume $co\ arg$)
    and $rehandle\ k\ arg$ = $handler\ eff\ continue\ effh\ (\lambda\_.k\ arg)$
    and $handle\ r$ =
      match $r$ with
      | $Eff\ eff'\ v$                when $eff' = eff$ → $effh\ v\ continue$
      | $Eff$ _ _                                  → yield ($Resend\ r\ continue$)
      | $Resend\ (Eff\ eff'\ v)\ k$   when $eff' = eff$ → $effh\ v\ (rehandle\ k)$
      | $Resend\ effv\ k$                          → yield ($Resend\ effv\ (rehandle\ k)$)
      | _                                          → $vh\ r$
    in $continue$ nil
  in $handler$
```

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

The term after translation contains the function *handler* defined in Figure 3, which works as follows: `handler` makes a thunk from $(\lambda\_.\cdots\cdots)$, defines three functions *continue*, *rehandle* and *handle*, and then evaluates *continue* `nil`. *continue* passes *arg* to *co*, `resume`-s it, and passes the return value to *handle*.

```
$M$ = let $h_1$ = handler $C_1$
          (val $v \to v$) $((x, k) \to kx)$ in
      let $h_2$ = handler $C_2$
          (val $v \to v$) $((x, k) \to kx)$ in
      let $h_3$ = handler $C_3$
          (val $v \to v$) $((x, k) \to kx)$ in
      with $h_3$ handle
      with $h_2$ handle
      with $h_1$ handle
        let $a$ = perform ($C_1$ 10) in
        let $b$ = perform ($C_1$ 13) in
        let $c$ = perform ($C_3$ 17) in
        $a + b + c$
```

Fig. 4: Example program in λ_{eff}

```
$[\![M]\!]$ = let $h_1$ = let $vh_1 = \lambda v.\ v$ in
                       let $\mathit{effh}_1 = \lambda x.\ \lambda k.\ k\ v$ in
                       $\mathit{handler}\ C_1\ vh_1\ \mathit{effh}_1$ in
          let $h_2$ = let $vh_2 = \lambda v.\ v$ in
                       let $\mathit{effh}_2 = \lambda x.\ \lambda k.\ k\ v$ in
                       $\mathit{handler}\ C_2\ vh_2\ \mathit{effh}_2$ in
          let $h_3$ = let $vh_3 = \lambda v.\ v$ in
                       let $\mathit{effh}_3 = \lambda x.\ \lambda k.\ k\ v$ in
                       $\mathit{handler}\ C_3\ vh_3\ \mathit{effh}_3$ in
          $h_3\ (\lambda\_.\ h_2\ (\lambda\_.\ h_1\ (\lambda\_.$
            let $a$ = yield ($\mathit{Eff}\ C_1$ 10) in
            let $a$ = yield ($\mathit{Eff}\ C_1$ 13) in
            let $a$ = yield ($\mathit{Eff}\ C_3$ 17) in
            $a + b + c$ )))
```

Fig. 5: Example after translation

handle splits the process from the return value of `resume` according to the equivalence of tags and effect labels.

When *continue* is evaluated by passing *nil*, *Eff* C_1 10 is `yield`-ed first in the handled expression and caught by the innermost handler h_1. In this case, since it has an *Eff* tag and h_1 can handle C_1, the first pattern of *handle* matches it. *effh* is applied to the effect's argument 10 and a continuation. By passing *continue* as the continuation, the computation of a handled expression can be resumed, which is suspended at the yielded position. And since *continue* passes the return value of `resume` to *handle*, the effect can be handled by the same handler again. So a is bound to 10. When *Eff* C_1 13 is yielded in the continuation resumed by the handler, it is processed by h_1 again in the same way, and b is bound to 13.

When the effect C_3 is invoked, h_1 catches the effect first. h_1 can't handle C_3, so the second pattern of *handle* matches. The effect is sent to a handler one step outside, and the effect is processed by that handler. As with invoking an effect, an effect is re-sent to an outside handler by using a `yield`. At this time, the tag *Resend* wraps the effect and a continuation to indicate resending. Then, as in the first pattern, pass *continue* as a continuation.

The resent C_3 is captured at h_2. Since it has *Resend* tag and h_2 can't handle C_3, the fourth pattern of *handle* matches. As in the second pattern, it uses *yield* to re-send the effect to an outside handler. At this time, *rehandle k* is wrapped by *Resend* tag as a continuation. *rehandle* is a function that 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*, *handle* calls *yield*, so control is exited from one coroutine. In the third and fourth clauses, we could write $\lambda arg.handle\ (k arg)$ instead of *rehandle* k, if only to manipulate the return value of the continuation. In this case, the layers of the coroutines would decrease, and eventually, we would get an error calling `yield` outside of coroutine. Therefore *rehandle* encapsulates the expression with coroutine internally and avoid to decreasing the layer of coroutines.

The effect resent again is captured by h_3. It has *Resend* tag and h_3 can handle C_3, so the third pattern of *handle* matches. Same to the fourth pattern, *rehandle* k is passed to *effh* as a continuation. Then it returns 17 to the handled expression, and c is bound to 17.

The handled expression results in 40. Then h_1 receives it, and the fifth wild-card pattern of *handle* matches the untagged value, and the value is passed to the value handler. Same to the h_1, h_2 and h_3 receive the value and pass to the value handler. Finally the entire expression returns 40.

Although our translation looks complicated, we emphasize that our translation is compositional and local, syntax-directed, and does not rely on higher-order stores or other fancy features, but need only basic functionality of asymmetric coroutines. With this simplicity, several programmers have already ported our translation to other languages than Ruby and Lua.

3.4 Macro-expressible Translation

We 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 discusses 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. 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 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.

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[11], 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 Figure 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. (end of proof sketch)

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 one, 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 Section 3. Since the translation is macro-expressible, we can realize our implementation as a simple library. [Kwhr: fix 31] Our implementations are compact. The

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

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 we will address 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.

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 needs the uniqueness of effect instances, 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 besides AEH, it will cause a serious problem, since other effects 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 implementation of) AEH.

[Kwhr: fix 22 and 23] This problem can be solved as follows, thanks to the expressivity of AEH. See the following code.

```
local Yield = inst()

local yield = function(v)
  return perform(Yield, v)
end

local create = function(f)
  return { it = f, handled = false }
end

local resume = function(co, v)
  if co.handled then
    return co.it(v)
  else
    co.handled = true
    return handler({
      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 above is an implementation of 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 Figure 3. So we define `Yield` effect (line 1) and `yield` function (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 `Yield` effect 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` (line 19 and 20). Since we provide a deep handler, it is not necessary to set the handler multiple times. The tag `handled` is to assert if the function is handled by the handler or not (line 12). The function `resume` checks the flag; if the flag is off, `resume` turns on the flag and runs the function with the handler. Otherwise, `resume` runs the function only.

Although we believe that the above technique may be used in other computation effects, it is left for future work to combine them with algebraic effect and handlers without big downgrade of performance.

5 Evaluation

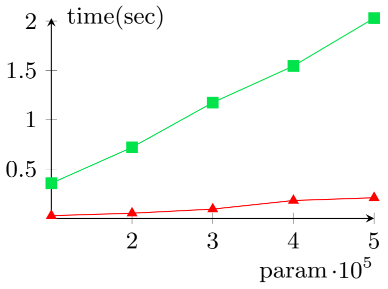
We have conducted experiments on microbenchmark using our library in Lua, and implementation in Lua based on free monads [23], and compare their performance. All the code for the benchmark is publicly available in the GitHub repository[12]. In the following figures, the symbol ▲ represents the result of our library, and 🟩 does of the free-monad based implementation. One of the benchmarks compares to native coroutines of Lua and indicates the result as the symbol $\star$ in a graph. The experiments have been conducted on the environment in Table 1.

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, and recursively runs a simple computation consisting of one-layer one-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

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

Table 1: Environment for Benchmark

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

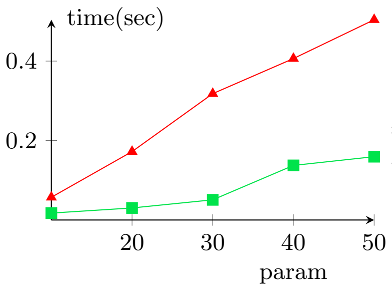


param ($10^5\times$)	1	2	3	4	5
free	0.3560	0.7199	1.1741	1.5449	2.0303
ours	0.0273	0.0521	0.0939	0.1811	0.2090

Fig. 6: Result of `onestate` benchmark

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.

In the next experiments in Figure 7, the benchmark program iterates `count` function 3,000 times in deeply nested handlers. The parameter in the table

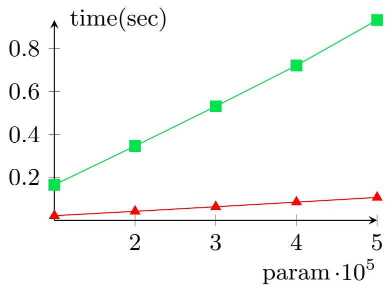


param ($10\times$)	1	2	3	4	5
free	0.0174	0.0303	0.0508	0.1374	0.1594
ours	0.0572	0.1722	0.3181	0.4064	0.5041

Fig. 7: Result of `multistate` benchmark

corresponds to the number of nested handlers/coroutines, hence 50 (the rightmost column) is already a rather unrealistic situation, but we included this experiment as an extreme. 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 Figure 3, so it degrades the performance.

In the next experiment, the function `looper` performs algebraic effects in the iteration of the `for` loop, where the number of iteration is given as a parameter shown in the table of Figure 8. The benchmark program invokes an effect in a

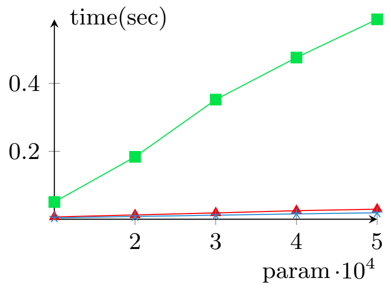


param ($10^5\times$)	1	2	3	4	5
free	0.1650	0.3455	0.5304	0.7204	0.9319
ours	0.0215	0.0416	0.0630	0.0844	0.1064

Fig. 8: Result of `looper` benchmark

`for`-loop and sets a handler out of the loop to catch the effect. 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 be able to eliminate all or part of this overhead.

Figure 9 shows the result of the benchmark, which solves the same-fringe problem [11] by using algebraic effects and coroutines. [Kwhr: fix 54] The problem



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

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 implement coroutines to solve it, by algebraic effects with free monad, and our library, described in Section 4. We also implement the 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.

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 can catch the effect invocation even during the execution of the continuation.

In the literature, there has been a discussion on deep vs *shallow handlers* [12], and it has its own merits. We have also implemented the shallow handler with coroutines shown in Figure 10. The idea is simple; after a handler catches an effect, it always resends any effects to the outer handler. We have explained the role of *rehandle* in Figure 3 that it encapsulates the continuation with a coroutine to adjust the layer of coroutines, and rehandles the effect invocation in the continuation. In the shallow setting, it is also necessary to reset the number of the layer of coroutines, which might degrade the performance. On the other hand, rehandling is not needed because it is shallow.

One-shot Continuations It should be noted that we are not the first to study the one-shot variant of control operators. Bruggeman et al. give an one-shot control operator *call/1cc* with the observation that almost continuations are run at most once [5]. When a compiler knows a continuation can be run at most once, it can generate more sophisticated code. They state that, by replacing *call/cc* using continuations at most once for *call/1cc*, program can be run with less memory consumption and higher performance. Berdine et al demonstrate that many control abstractions can be translated into typed CPS including one-shot continuations, with linear-types [2].

[Kwhr: fix 55] James and Sabry stated that the *yield* operator for generator, which is a restricted variant of coroutines and can be found in various languages, is one-shot delimited continuations [13]. They also defined a generalized *yield* operator which has multi-shot continuations and show the connection between it and the 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]. They provide one-shot continuations due to the performance problem, and if multi-shot continuations are needed, they allow explicit copy for continuations.

```
  $[\![$handler$^\dagger$ $\mathit{eff}$ (val $x \to e_v$) $((x,k) \to e_{\mathit{eff}})]\!] =$
                                          let $\mathit{eff} = [\![\mathit{eff}]\!]$ in
                                          let $\mathit{vh} = \lambda x.[\![e_v]\!]$ in
                                          let $\mathit{effh} = \lambda x\ k.[\![e_{\mathit{eff}}]\!]$ in
                                          $\mathit{handler}^\dagger\ \mathit{eff}\ \mathit{vh}\ \mathit{effh}$
where $\mathit{handler}^\dagger =$
  let rec $\mathit{handler}\ \mathit{eff}\ \mathit{vh}\ \mathit{effh}\ \mathit{th} =$
    let $\mathit{co}$ = create $\mathit{th}$ in
    let rec $\mathit{continue}\ \mathit{arg} = \mathit{handle}$ (resume $\mathit{co}\ \mathit{arg}$)
    and $\mathit{rehandle}\ k\ \mathit{arg} = \mathit{handler}\ \mathit{eff}\ \mathit{continue}\ \mathit{effh}\ (\lambda\_.k\ \mathit{arg})$
    and $\mathit{continue}_0$ = resume $\mathit{co}$
    and $\mathit{rehandle}_0\ k$ = resume (create $k$)
    and $\mathit{handle}\ r =$
      match $r$ with
      | $\mathit{Eff}\ \mathit{eff}'\ v$             when $\mathit{eff}' = \mathit{eff} \to \mathit{effh}\ v\ \mathit{continue}_0$
      | $\mathit{Eff}$ _ _                              $\to$ yield ($\mathit{Resend}\ r\ \mathit{continue}$)
      | $\mathit{Resend}\ (\mathit{Eff}\ \mathit{eff}'\ v)\ k$ when $\mathit{eff}' = \mathit{eff} \to \mathit{effh}\ v\ (\mathit{rehandle}\ k)$
      | $\mathit{Resend}\ \mathit{effv}\ k$                  $\to$ yield ($\mathit{Resend}\ \mathit{effv}\ (\mathit{rehandle}_0\ k)$)
      | _                                  $\to \mathit{vh}\ r$
    in $\mathit{continue}$ nil
  in $\mathit{handler}$
```

Fig. 10: translation from shallow handlers to coroutines

Free monad We have already compared our with with free-monad based implementations of algebraic effects. On the positive side, it gives a systematic and elegant implementation for various effects. Its downside is it has significant overhead in performance. We also point out that our embedding-based implementation does not interfere with surface languages, while free-monad based implementations force a programmer to use monadic style, which is good for some programmers, but is not for others. With our implementation, the surface language with algebraic effects and handlers can be presented in direct style or monadic style.

7 Conclusion

We have presented a novel embedding technique for algebraic effects and handlers into asymmetric coroutines, and shown translation from the former to the latter as simple, direct, syntax-directed compositional translation. Compared with other embeddings or other ways, our technique can apply to many languages which have coroutines due to the simplistic nature of our embedding. 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 uses the rest of the coroutine thread as a continuation, and the status of the coroutine cannot be copied, so the limitation exists that a continuation can be executed at most once. One-shotness is a dynamic property, and its static approximation, linearly used (delimited) continuations, or linear continuation-passing style, are the target of active research in the past. 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 state 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[13], which allows programmers to build components with side-effects modularly. Abramov pointed out the relevance between Hooks and algebraic effects in his blog post[14], and we think that investigating this relationship based on our work is promising.

References

1. Bauer, A., Pretnar, M.: Programming with Algebraic Effects and Handlers. Journal of Logical and Algebraic Methods in Programming **84**, (03 2012)
2. Berdine, J., O'Hearn, P., Reddy, U., Thielecke, H.: Linear Continuation-Passing. Higher-Order and Symbolic Computation **15**, 181–208 (09 2002)
3. Brachthäuser, J., Schuster, P.: Effekt: extensible algebraic effects in Scala (short paper). pp. 67–72 (10 2017)
4. Brachthäuser, J., Schuster, P., Ostermann, K.: Effect handlers for the masses. Proceedings of the ACM on Programming Languages **2**, 1–27 (10 2018)
5. Bruggeman, C., Waddell, O., Dybvig, R.: Representing Control in the Presence of One-Shot Continuations. vol. 31, p. (02 1970)
6. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming. p. 151–160 (1990)
7. Dolan, S., White, L., Madhavapeddy, A.: Multicore OCaml. In: OCaml Users and Developers Workshop (2014)

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

8. Felleisen, M.: On the Expressive Power of Programming Languages. In: Selected Papers from the Symposium on 3rd European Symposium on Programming. p. 35–75. ESOP '90, Elsevier North-Holland, Inc., USA (1991)
9. Felleisen, Matthias and Daniel P. Friedman: 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, https://doi.org/10.1017/S0956796819000121
11. Gabriel, R.P.: The Design of Parallel Programming Languages, p. 91–108. Academic Press Professional, Inc., USA (1991)
12. Hillerström, D., Lindley, S.: Shallow Effect Handlers. In: Asian Symposium on Programming Languages and Systems. pp. 415–435. Springer (2018)
13. James, R., Sabry, A.: Yield: Mainstream Delimited Continuations. p. (01 2011)
14. Kammar, O., Lindley, S., Oury, N.: Handlers in Action. vol. 48, pp. 145–158 (09 2013)
15. Kiselyov, O., Ishii, H.: Freer Monads, More Extensible Effects. ACM SIGPLAN Notices **50**, (03 2015)
16. Kiselyov, O., Sivaramakrishnan, K.: Eff Directly in OCaml. Electronic Proceedings in Theoretical Computer Science **285**, 23–58 (12 2018)
17. Leijen, D.: Algebraic Effects for Functional Programming. Tech. rep., Technical Report. 15 pages. (2016)
18. Leijen, D.: Implementing Algebraic Effects in C. pp. 339–363 (11 2017)
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, ICFP 2011, Tokyo, Japan, September 19-21, 2011. pp. 81–93. ACM (2011). https://doi.org/10.1145/2034773.2034786, https://doi.org/10.1145/2034773.2034786
20. Moura, A.d., Ierusalimschy, R.: Revisiting Coroutines. ACM Transactions on Programming Languages and Systems **31**, (07 2004)
21. Plotkin, G., Power, J.: Algebraic Operations and Generic Effects. Applied Categorical Structures **11**, 69–94 (02 2003)
22. Plotkin, G., Pretnar, M.: Handling Algebraic Effects. Logical Methods in Computer Science **9**, (12 2013)
23. Pretnar, M., Saleh, A.H., Faes, A., Schrijvers, T.: Efficient compilation of algebraic effects and handlers. CW Reports, volume CW708 **32** (2017)

A Semantics of λ_{eff}

A.1 Helper functions

We introduce three helper functions for semantics in Figure 11: [Kwhr: fix 16]

$$
\begin{aligned}
\mathit{split}\left(\left((\texttt{with}\ w\ \texttt{handle}\ \Box)^{\mathit{eff}} :: K\right), \mathit{eff}\right) &= \left([\,], (\texttt{with}\ w\ \texttt{handle}\ \Box)^{\mathit{eff}}, K\right) \\
\mathit{split}\left((F :: K), \mathit{eff}\right) &= \left(F :: K', F', K''\right) \\
&\quad \text{where } F \neq (\texttt{with}\ w\ \texttt{handle}\ \Box)^{\mathit{eff}} \\
&\quad \text{and } \left(K', F', K''\right) = \mathit{split}\left(K, \mathit{eff}\right) \\
(\!| F :: K |\!) &= \lambda x.\ (\!| K |\!)\ F\,[x] \\
(\!|\ [\,]\ |\!) &= \lambda x.\ x \\
K * E &= \texttt{clos}\,(\lambda x.(\!| K |\!)\ x, E)
\end{aligned}
$$

Fig. 11: Helper Functions for Semantics

$\mathit{split}\,(K, \mathit{eff})$ returns a triple (K_1, F, K_2) where F is the frame that handles the effect named by *eff*, and $K = K_1 :: [F] :: K_2$ holds. [Kwhr: fix 31] If more than one frame can handle the effect *eff*, the first one is selected, and if none have the named effect, the result is undefined. $(\!| K |\!)$ converts a stack K to a continuation in functional form. $(K * E)$ creates a closure with a stack frame K and an environment E.

A.2 Small-step semantics

Figure 12 defines the small-step, call-by-value, left-to-right semantics ($\longrightarrow_{\mathrm{eff}}$) in the CEK-machine style [9].

In the rule LOOKUP, $E\,(x)$ is the value associated with the variable x in the environment E. The rules PUSHLET, BIND, and CLOSE, PUSHAPP, PUSHARG, and APP are [Kwhr: fix 17] standard. The rest of the rules are the one for algebraic effects and handlers. The rules PUSHWITHHANDLE and CLOSEHANDLER push or pop evaluation contexts to the stack. The rule HANDLE manipulates a with-expression `with` h `handle` e: if h evaluates to a handler value, then e is going to be evaluated under this handler. The rule PUSHPERFORM pushes the frame of `perform`-ing an effect *eff* to the stack. The rules HANDLEPERFORM and HANDLEVALUE are the key rules for algebraic handlers. In the rule HANDLEPERFORM, the code component is a value w. Hence, the first frame in the stack `perform` *eff* $\Box$ is retrieved and evaluated. Then we look for a handler whose name is *eff* in the stack K, and if we find it, we use the handler to cope with this effect where formal parameters y and k are bound to the value w and the delimited continuation K' under environment E. We adopt the *deep* handlers, hence the handler $(\texttt{with}\ w_h\ \texttt{handle}\ \Box)^{\mathit{eff}}$ remains in the stack after this step. The rule HANDLEVALUE is used when the handled expression does not invoke an effect and returns a value w. Then the value handler $(\texttt{val}\ x \to e_v)$ is used, and the handler is eliminated from the stack after this step.

B Semantics of $\boldsymbol{\lambda_{ac}}$

Auxiliary functions Figure 13 defines two auxiliary functions for pattern matching of λ_{ac}. $FV_p\,(pat)$ is the set of free variables in *pat*. $matchable\,(v, pat)$ is a predicate to assert that,given a value v and a pattern *pat*, the value matches the pattern. The operator $\oplus$ concatenates two stores and $\emptyset$ is an empty store. The function *genstore* creates a new store which consists of pairs of a variable and a value (which consists of constructors). For example, by calling *genstore* with the arguments $Resend\ \ (Eff\ \ w\ \ v)\ \ u$ (for some values w, v and u), and a nested pattern $Resend\ \ (Eff\ \ y\ \ x)\ \ k$, we get a new store $\emptyset\,[y \leftarrow w, x \leftarrow v, k \leftarrow u]$.

Small-step Semantics Figure 14 shows the operational semantics of λ_{ac} by the transition ($\longrightarrow_{\text{ac}}$) of the state $\langle e, \theta\rangle$, an expression e and a store θ. $dom\,(\theta)$ is the domain of θ, and $\theta\,(x)$ is a value associated with the variable x. Note that even if $\theta\,(l) =$ `nil`, we include l in $dom\,(\theta)$. In those cases such as introducing a variable or a label (App, Let, LetRec, Create, Match, and MatchWhen), we identify α-equivalent terms and assume that we rename bound variables appropriately for substitution to be defined at any time. The distinctive pattern _ is similar to a variable but generates no binding after pattern matching, so we allow _ to be overwritten. The rules contain those for variable lookup (Lookup), function application (App), `let`, and `let rec` (Let and LetRec). The function Create is to make a new coroutine. It creates a fresh label l, binds the coroutine to l, and returns its label to the context C. The function Resume produces a labelled expression, an application $\theta\,(l)\ \ v$ with a label l. $\theta\,(l)\ \ v$ is what finds the body corresponding to the label l from θ and apply v. The created labelled expression $l : \theta\,(l)\ \ v$ expresses the computation in the coroutine labelled by l. To prevent the rest of the coroutine from being referred, the rule Resume invalidates the associated

value by setting it to `nil`. The function Yield suspends the current computation of a coroutine and returns to the parent coroutine with an argument. Since the target calculus represents asymmetric coroutines, a coroutine can be a parent of another coroutine by resuming it. [Kwhr: fix 29] The rule have an assumption that C_2 does not have any labelled expresses. The assumption indicates that C_2 is the innermost active coroutine. The function LabelledReturn transfers the result of the computation v in the coroutine l to its caller. The functions EqT and EqF compare two effect operations. The operator $=_{\mathit{eff}}$ judges whether two given effects are the same. The functions Match and MatchWhen are for pattern-matching. The second rule applies when $K\ \overrightarrow{v}$ matches a pattern, and the match case has a guard c. This rule transforms the guard to another match expression, with assigning the values to the corresponding variables in the pattern. The assignment may affect pattern variables in the guard c. If a guard returns `True`, pattern matching is successful, and the body of the `True` clause is evaluated; otherwise, we go to match against the rest of the patterns.

C Reply to reviewers

```
Dear reviewers of an earlier version of this paper,

Thank you very much for your careful reading and helpful comments.
We do appreciate all your comments very much.  In this revision,
we did our best to address the issues you raised. We have solved
most issues but unfortunately we have not been able to manage all,
and here we show how and what we did for your comments. (To avoid
clutter, we omitted responses for small literal errors.)

Best regards,
Satoru Kawahara and Yukiyoshi Kameyama


 >>---------------------- REVIEW 1 --------------------
 >>Cons:
 >>  - No comparison is made with de Moura and Ierusalimschy's earlier work,
 >>    which already shows that one-shot continuations can be encoded into
 >>    asymmetric coroutines.
 >>    In fact, perhaps the encoding presented here could be presented as the
 >>    composition of a simpler encoding (from multiple effects to a single
 >>    effect) with de Moura and Ierusalimschy's encoding. (See below.)

It is true that we can give a translation from a calculus with one-shot
control operators to a calculus with asymmetric coroutines, by such a
composition. The merit of our translation is that it is given as
a simple, local translation (de Moura and Ierusalimschy's translation
uses mutable state which may store higher order functions.)  This property
is made explicit as a 'macro-expressible' translation in the sense of
Felleisen (Sect. 3.4 in the revised paper). Although it is a very simple
property, we think that it is sufficient to claim that our translation is
simple.  The property also lets us implement the translation as a simple
library (which does not need any external resource).

 >>  - The encoding is not well-explained.

We are sorry for this. In fact, we did not give explanation in the
earlier version.  The revised version contains a concrete example of
translation (encoding) and we gave a detailed explanation based on the
concrete example. (Sect. 3.3)

 >>MAJOR GENERAL COMMENTS
 >>
 >>
 >>>  It seems to me that coroutines, as presented in this paper (the calculus
 >>>  lambda_{ac}), are a mix of delimited control and dynamically-allocated mutable
 >>>  state. Because of this, instead of implementing delimited control in terms of
 >>>  coroutines, as proposed in the present paper, it would seem rather more
 >>>  natural to me to explain (or implement) coroutines in terms of delimited
 >>>  control and mutable state. I can see that the encoding proposed in this paper
 >>>  is useful in situations where the host language has coroutines (e.g. Lua), but
 >>>  otherwise this encoding seems to be going in the wrong direction. In a
 >>>  language where mutable state and (one-shot) effect handlers are primitive,
 >>>  coroutines can be programmed as a library. (See e.g. Section 4 of "Effect
 >>>  Handlers for the Masses", by Brachthäuser et al.) Isn't that a more pleasing
 >>>  approach?

Thank you for your insight. We agree that the converse direction is worth studying,
but we still do believe the present direction (from algebraic effects and handlers to
corouines) is interesting, as it gives an implementation of the former in a huge number
of programming languages. (The cretin Henry in Wikipedia has a large list of
programming languages with native support for coroutines; although we have not
examined the list in detail, as to whether they support stackful asymmetric coroutines
which is needed for our work, we can surely say that the direction to coroutines
is worth studying.) In the revised version, we modified the introduction (Sect. 1)
to mention these facts briefly.

 >>>  p.2, I am confused by the sentence "However, as we will see in this paper,
 >>>  coroutines are as expressive as one-shot delimited control-operators, a
 >>>  restricted control operator that is allowed to invoke a delimited continuation
 >>>  at most once [13]." Without the citation at the end, this sentence seems to
 >>>  suggest that the equivalence between coroutines and one-shot delimited control
 >>>  is a contribution of this paper. The abstract of the paper suggests the same
 >>>  thing. Yet, once one looks up the citation [13], one discovers that this
 >>>  equivalence has already been pointed out by de Moura and Ierusalimschy. As a
 >>>  result, at this point, I am confused, and do not know exactly how the present
 >>>  paper improves on (or distinguishes itself from) de Moura and Ierusalimschy's
 >>>  paper. This should be clarified!

The sentence was our mistake; we do not claim that we are the first to
make this equivalence (or connection), and we have removed the phrase "as we will
see in this paper,".

 >>> Although the encoding (Figure 5) is relatively compact, the key part (the
 >>> definition of "handler") is still 10 lines of complex code, and the
 >>> explanation on page 12 does not really help; it is a paraphrase of the code.
 >>> I suspect that part of the complexity has to do with the fact that the source
 >>> calculus has multiple named effects (like "Get", "Set", "Defer", etc.) and it
 >>> is possible to perform an effect that is handled by a handler which is *not*
 >>> the nearest handler; whereas the target calculus does not have a comparable
 >>> mechanism ("yield" transfers control to the nearest mark on the stack). So,
 >>> I wonder if the encoding could be viewed as the composition of two separate
 >>> (simpler) encodings, namely:
 >>>
 >>>  - an encoding of a calculus with multiple (named) effects
 >>>    into a calculus with a single effect; and
 >>>  - an encoding of this calculus with a single effect
 >>>    into a calculus with coroutines.
 >>>
 >>> The first encoding would take care of comparing effect labels for equality,
 >>> wrapping effects in "Resend", etc., while the second encoding would take care
 >>> of encoding "perform" into "yield". Does that make sense? Is this possible?

This perfectly makes sense, but, the definitions in this way are (conceptually
much simpler, but) not that simple, and we need to explain two non-trivial
translations, which needs huge space. Also if our translation is a composition
of two translations, which is less efficient the one given in this paper.
We will try again when we revise this paper in future (e.g. writing a journal version).

 >>> Furthermore, the second encoding above might be essentially identical to the
 >>> encoding proposed by de Moura and Ierusalimschy. (They encode a calculus with
 >>> a "spawn" operator, which looks analogous to shift/reset, into a calculus with
 >>> coroutines.) If that were the case, then the contribution of this paper would
 >>> have to be re-explained in a different way.

As we wrote above, one of the major merits of our translation is that it is
a macro-expressible translation, which distinguishes itself from the earlier
work by de Moura and Ierusalimschy (though the translations share the same spirit.)

 >>MINOR GENERAL COMMENTS

 >>> Is there a proof that this encoding is correct? (A machine-checked proof would
 >>> be ideal.) I would not be interested in reading the details of the proof, but
 >>> I would be interested in reading the main invariants that explain why the
 >>> encoding works, and it would be good to know that a proof exists.

Unfortunately the precise (e.g. formalized in Coq) proof does not yet exist, though
it is a relatively routine work to carry out the proof.

 >>> If the source and target calculi were equipped with (standard) type systems,
 >>> would the encoding be type-preserving? It would be worth answering this
 >>> question (if the answer is obvious) or explicitly deferring it to future work
 >>> (if nonobvious).

Well, it *is* an interesting future work and we mention it in the conclusion.
```

The type system of asymmetric coroutines is not so trivial -- there exist a few such proposals, but as long as we have examined much earlier, none of them is sufficiently strong (even a very simple typed calculus for shift and reset cannot be translated to any of them in a type-preserving way.)

```
 >>DETAILED COMMENTS
 >>> p.1, "Implementations based on stack manipulation are used in JVM bytecode and
 >>> C implementations". Please provide citations at this point. Same request about
 >>> "OCaml implementation and one of Scala implementations". (In fact, the
 >>> required citations are given at the end of the first paragraph, but it was not
 >>> evident to me at first that these citations were specifically concerned with
 >>> the implementation of effect handlers; I initially thought that were generic
 >>> citations about the programming languages Haskell, OCaml, etc.)
```

Thank you for the comment. We have added citations and also rewritten the sentences.

```
 >>> p.1, "it highly depends": what does "it" refer to?
 >>> p.1, "which needs deep insight on language processors": who needs deep insight?
 >>> The runtime system? What does that mean? And what is a "language processor"?
 >>> I don't think this terminology is standard.
```

We rephrased this to "an implementer needs deep insight on its internal structure."

```
 >>> p.1, the English in paragraph 2 is not great. It would be good if the paper
 >>> could be proof-read by someone who is proficient in English.
```

We are sorry. We tried to revise it based on the comments.

```
 >>> p.2, "implementing delimited control-operators in a language is non-trivial":
 >>> isn't just as difficult to implement delimited control operators and to
 >>> implement effect handlers? Each feature can encode the other. (Assuming an
 >>> appropriate control operator is chosen.) (See e.g., Forster et al, "On the
 >>> expressive power of user-defined effects: Effect handlers, monadic reflection,
 >>> delimited control".)
```

Yes, you are right and we have removed the sentence.

```
 >>> p.2, "We will show that using coroutines to implement algebraic effects and
 >>> handlers have another merit: a programmer does not have to think about what
 >>> data structures are used to embed algebraic effects." This sentence is unclear
 >>> to me in several ways. The authors write that this implementation scheme has a
 >>> "merit": in comparison with what other implementation scheme? And why would
 >>> the programmer have to think about data structures? One might hope that,
 >>> regardless of how effects are implemented, the programmer does not have to
 >>> think about it.
```

We have removed the sentence.

>>> p.2, "Although implementations of algebraic effects using coroutines already
>>> exist such as 10 11 12, each one has such problems as it does not provide a
>>> first-class continuation to the user, or is rather complicated." Please
>>> clarify which implementation has which deficiency. Also, please clarify your
>>> criticisms. "Not providing a first-class continuation to the user" sounds like
>>> a critical problem: if that is the case, then this library *cannot* be called
>>> an implementation of algebraic effects, can it? "Being rather complicated" is
>>> somewhat vague; could you be more specific?

We have removed the original sentences, and rewritten the whole paragraph.

>>> p.3, "We show a new embedding". The word "new" suggests that there already
>>> exist known encodings of algebraic effects in terms of coroutines in the
>>> literature? Is this the case? If so, please cite them explicitly. If not,
>>> then remove the word "new". (I suggest writing "We give an encoding of
>>> one-shot algebraic effects and handlers into asymmetric coroutines.")

Yes, there is no such encoding, and we have removed 'new'.

>>> p.3, "We do not use continuation-passing style nor user-level control
>>> operators. As a consequence, our embedding is more performant in many cases
>>> than other embeddings, including the one with free monads." This sentence
>>> could be clarified to indicate exactly which other encodings you are comparing
>>> against.

We were comparing it with free-monad based embedding, and we rephrased the sentence.

>>> Section 2.2, this material is standard. A citation of the original paper where
>>> this is described would be welcome. Also, I am not sure how this example adds
>>> value to the paper.

Well, we think that many readers are not familiar with even syntax of Lua etc.
and an easy introductory example is necessary for a technical paper.

>>> Section 2.3, this example seems a little underwhelming (I mean, not very
>>> convincing), because it seems clear that "defer" can be implemented using
>>> just first-class functions and references.

Thank you. We tried to fix the problem, but unfortunately due to lack of space,
finally we had to eliminate the example completely.

>>> The placement of Figure 2 is not great. There is one line of text below it.

```
Thank you, fixed it.

 >>> p.9, what does "more or less standard" mean?

Removed the phrase "more or less".

 >>> p.9, "We adopt the deep handlers". A citation of a paper that explains the
 >>> distinction between shallow and deep handlers would be welcome.

We have added one reference, though we have not found any good reference about this
discussion.

 >>> p.11, "Due to lack of space [...]". The reader who is not familiar with
 >>> coroutines needs at least an intuitive explanation of the meaning of
 >>> labeled expressions and the operations create, yield, resume. It would
 >>> be easy to make some room in the paper by removing Section 2.2 or 2.3.

We have revised the explanation of the informal semantics of coroutines.

 >>> In Figure 5, the various renamings of x to x' are just noise, as far as
 >>> I can see. Why not just keep the name x?

Yes, you are right, primes have been eliminated.

 >>> p.13, "our translation [...] does not rely on higher-order stores": the
 >>> operational semantics of coroutines does rely on a store (a mapping of
 >>> labels l to lambda-abstractions), so this statement is questionable.

De Moura and Ierusalimschy's translation (not semantics) does use higher-order
stores.

 >>> p.14, "Representing coroutines and other effects using algebraic effects and
 >>> handlers is possible, but tedious if the language has coroutines from the
 >>> beginning." I am not sure if "tedious" is the right word. It does not sound
 >>> difficult to encode coroutines in terms of effects (the encoding is known),
 >>> but this adds a lot of overhead, since one has to go through two layers of
 >>> encoding instead of using native coroutines.

Yes, we have removed the word 'tedious' and rewritten the whole sentence.

 >>> p.14, "This problem can be solved as follows [...]." I do not understand what
 >>> is going on here. The previous sentence suggested that encoding coroutines in
 >>> terms of effects is of course possible, but very costly. The text that follows
 >>> repeats that it is possible to encode coroutines in terms of effects, and gives
 >>> the encoding. How does that "solve" the problem? The encoding remains very
 >>> costly.

Since we have removed the sentence just before this one, we think the resulting
text makes sense.

 >>> p.15, "Although we know several solutions to this direction, clearly we need
 >>> to do more [...]". This sentence is very unclear. Which solutions do you know,
 >>> and why are they not satisfactory? Only one solution has been shown.

We have rewritten it to
  Although we can combine our library and coroutines by re-
  implementing coroutines as above, whenever we want to use other native effects,
  we must deal with them.

 >>> p.19, "Some of them have implemented our translation for other languages such
 >>> as JavaScript and Rust". This seems puzzling: does Rust have coroutines? How
 >>> are they implemented? As far as I know, they are not a primitive construct in
 >>> Rust.

Thank you. We rewrote the footnote to
   Since Rust does not have coroutines, we use Future instead.

 >>> p.20, "a new store which consists of pairs of a variable and a value".
 >>> A store usually maps *locations* to values. An environment usually maps
 >>> *variables* to values. It is confusing to see these notions apparently
 >>> mixed up here. What is going on?

The paper 'Revisiting coroutines' uses this terminology (which might be somewhat
non-standard) and we want to follow their way as long as coroutines are concerned.

 >>> p.21, "θ is a partial map from variables or labels to values". This is also a
 >>> bit confusing. A map of variables to values is an environment. However, here,

Yes, but again we followed the paper 'Revisiting coroutines'.

 >>> The authors adopt environment-based semantics, as opposed to
 >>> substitution-based semantics, for the source and target calculi.
 >>> Why is that? Could this decision be discussed?

We think environment-based semantics is also natural (and slightly more
practical).

 >>> p.22, the reduction rule (Yield) seems non-deterministic, as there could be
 >>> several marks "l:" on the stack. Should one add a side condition stating that
 >>> the context C_2 contains no marks? (See the distinction between C and C' in
 >>> de Moura and Ierusalimschy's semantics.)

The rule have an assumption that C2 has no labeled expresses, then C2 must
be the innermost active coroutine.

 >>>done p.4, line 2 begins with a stray comma.
 >>>
 >>>done p.4, "with still handled" does not make sense.
 >>>
 >>>done p.11, "propositional": do you mean "compositional"?

Thank you for careful reading and spotting typos.

 >>---------------------- REVIEW 2 ---------------------

 >>I think this should be accepted for presentation. It's certainly in scope,
 >>describes a topic which is certainly part of a trend, and applies it in a
 >>less conventional (for this audience) setting. I'm not sure it's ready yet
 >>to be accepted for publication though. I'd like to see more discussion of
 >>the motivation and limitations. I assume, since Lua is a dynamically
 >>typed imperative language, the main motivation is to allow effects to be
 >>handled in different ways in different contexts? In which case, it'd be nice
 >>to be explicit about this and show examples to illustrate it. Another
 >>motivation for algebraic effects is to be explicit about the capabilities of
 >>a function in its type, so could your approach be adopted by a statically
 >>typed language to achieve this? And is it a serious limitation to have
 >>one shot continuations - it would prevent implementing non-determinism as
 >>an effect, for example.

We hoped so, but introduction type systems and making the translation type-preserving
are tricky, as the existing type system for coroutines are not expressive enough.
But this is an interesting future work, and we mention it.

 >>It's nice to see the benchmarks, but they are fairly small and I would also
 >>like to know what the overhead of the library is compared to writing
 >>effectful programs directly, and some discussion of whether that overhead
 >>is worth it. One example suggests the overhead is not too bad, but there
 >>has to be a strong enough motivation to explain why it's worthwhile.

We have added more figures and more benchmarks.  Although they are still
microbenchmarks, we think that the extreme-case examples show the small overhead
of our implementation.

 >>> p4 I'm not familiar with Lua, and I can guess what the notation
 >>>   [DivideByZero] means, but for this audience it might be worth explaining a
 >>>   bit more

We are sorry and we should not assume the familiarity with Lua. We have added more
explanations for Lua syntax and the code.

 >>> p6 Instead of giving this in Go, could you should how it’s used in your
 >>>   library instead?

We have tried to do so, but unfortunately we had to eliminate the last example
due to lack of space.

 >>> p7 "To implement the full functionality of defer..." - have you done this?

Yes, you can find it in our Github repository.

 >>> p16 Can you say in more detail what these benchmarks do? Rather than just
 >>>   saying which effect they use. Also for state in particular, it’d be nice
 >>>   to know the overhead over implementing state directly.

We have thoroughly revised the section for benchmark and performance evaluation,
which hopefully addresses your concern.

 >>>done p2 "thanks to that fact that" => "thanks to the fact that"

Thank you, fixed.

 >>> p3 What is ’inst()’? Something which creates a unique new effect label?

We explained it in the text.

 >>> p4 typo "catches the effect DivideByZero"
 >>> p8 "more than one frames have" => "more than one frame has"

Fixed.

 >>> p14 "The implementations are simple and easy to understand" - this is a
 >>>   strong claim, without showing any of the code here! Is there any part of it
 >>>   which you can present?

We have clarified the meaning (and rephrased the sentences).

 >>> p15 "the result of ours library" => "...our library"

Fixed.
```

>>---------------------- REVIEW 3 --------------------
>>One issue with the paper is that it doesn’t, from the start, state a clear, main goal.
>>The abstract states context, and the fact of what was implemented, but to what end?
>>As an advantage "modularity" is given. I think this could be made much stronger
>>pointing to the good performance results, and referring to the other aspects as
>>additional
>>advantages.

Thank you for the comment. Our goal is to make AEH available in MANY languages. (AEH for algebraic effects and handlers).
We think that algebraic effect and handlers are good abstraction for writing control effects, and implementer who write controlful programs in Lua would benefit from our work very much, as composing two effects realized by coroutines is not an easy task, while composing two algebraic effects is straightforward.
We have revised the abstract and introduction along with this line.

>>The formal presentation builds on the CEK machine, and adds rules for performing and
>>handling effects. The target of the translation is an existing calculus of asymmetric
>>coroutines. No formal correctness proof is given, so correctness of the translation
>>can’t be seen as a contribution. But the operational semantics of the CEK machine
>>with effects, and the translation look plausible. Also, the implementation has been
>>tested on a range of micro-benchmarks that are used for the performance results.
>>
>>Notably, the style of the translation should be applicable to a range of languages,
>>and the paper makes the point that it has been implemented not only in Lua and Ruby,
>>but also (by others) in more main-stream languages such as JavaScript and Lua.
>>
>>The performance results compare this translation with an existing free-monad
>>based implementation of effects and show sizable gains, and good scalability,
>>over increasing
>>numbers of iterations. However, there is an additional cost for the unlikely case of a
>>high number of nested handlers.

Thank you for your analysis, which spots merits and demerits of our current approach correctly.

>>> From the start, state the main goal: is it performance? or finding a translation
>>> that is modular, portable and high-performant at the same time.
>>> Too much space is spent on context here, IMHO, and could go into intro.

We have added a sentence. "This paper presents a translation from algebraic effects and handlers to asymmetric coroutines, which provides simple, portable and widely applicable implementation of algebraic effects."

```
 >>> Might mention that reasoning about effects is notoriously difficult,
 >>> and implementations
 >>> are therefore error-prone. Therefore a formalized implementation is very desirable.

We could have mentioned it, but since we do not really 'reason about' programs,
we hesitate to do so. But thank you for the very encouraging comment!

 >>> p2, top: refers to a range of implementations of effects, with pros and cons,
 >>> but doesn't
 >>> give references (at least to the main ones).

 >>> Highlight the "new embedding of alg effects and handlers" as main novelty.

We have revised the paper accordingly.

 >>> I'd expect more of a rationale why Lua and Ruby are used as languages.

Well, we think they are popular languages, and in particular, we believe that
Lua is the best language to study coroutines (but we cannot prove this claim).

 >>> Good to have several repos of implementations mentioned early in the paper.
 >>> Good summary of contribs: mention the main ones in the abstract already.
 >>> "conversion of \lambda_eff" is a bit vague: stick to the phrasing "translation"
 >>> throughout.

Yes, thank you, we revised it.

 >>> Before the first example I'd expect some words about Lua syntax. As it stands
 >>> it's hard
 >>> to read for someone not already familiar with the lang (main audience of TFP).
 >>> Suggest to number the lines in the examples to more easily refer to it.
 >>> Could use comments in the code to e.g. clarify that the 2 params to handler are the
 >>> (value) argument and the continuation.

We have added more explanations about the code with line numbers etc.

 >>> Some background on delimited continuations earlier in the paper would be useful.

It is hard to mention it within the limited space, but we have shown a concrete
example of a delimited continuation as a context.

 >>> What's the role of the value 0 passed to the continuation in case of a DivideByZero?

Not very much, but it is natural to pass some default value...
```

```
 >>> p5, bottom: in the text you talk about "binds s to v" and creating a thunk. Refer to
 >>> the lines in the code to link things up.

We did it.

 >>> last para: if you are saying the simplicity of implementation with your notation is
 >>> an advantage in itself, you should expand on it and mention this more prominently.

Our notion of simplicity is based on macro-expressiveness, which is now detailed
in Sect. 3.4.

 >>> Since you are building on the CEK machine, more context and concrete pointers
 >>> would be useful!

We have added a reference.

 >>> Giving intuitions for frames, possibly as "break-points" in the execution,
 >>> would be good
 >>> (together with general CEK machine background)

Here we use frames as just an element of a continuation (where a continuation
is a list, or a stack of frames.)  We have added a simple explanation for frames.

 >>> Fig 2: minor issue, but I don’t find the notation for these operations
 >>> very intuitive;
 >>>  maybe something more verbose such as "lookup" for // would be better?
 >>> Fig 3 and text: mention that CloseHandler and Handle rules are analogous to
 >>>  Close and App rules in the plain CEK machine.
 >>> I think some rules can be simplified: in CloseHandle you don’t refer to
 >>> the syntactic components of the handler h, so you don’t need to
 >>> expand its structure in the rule
 >>>  Might mention that HandlePerform and HandleValue correspond to the effect
 >>>  and no-efect
 >>>  cases respectively.

Well, we wanted to explain the semantics more but due to space limitation
we had to move the semantics section to appendix.

 >>>  One bigger issue I have with the notation is that ’::’ is used both for
 >>>  sequencing the list of frames in the continuations part, and
 >>>  for adding a binding to the environment.
 >>>  I think using more established notation for the latter, and
 >>>  thus separating the two
 >>>  operations would help here.
```

```
Here we use the symbol to mean the 'cons' operation for lists.

 >>> Fig 4: give in the text at least an intuition for the meaning of Eff, Resend etc.

We have added detailed explanation.

 >>> Fig 5 and text: the translation of the main case 'handler' is intricate,
 >>> but looks plausible;
 >>>  it would help to give some explanation on the structures Eff and
 >>>  Resume as well as the
 >>>  operations yield and resume that are used here. Maybe a simple
 >>>  presentation in a
 >>>  sequence-diagram style figure would help. I think at least this
 >>>  level of detail is needed
 >>>  as context for the last 2 paras on p12.

We have added a concrete example to show how the translation (and the translated
code) work.

 >>> A summary of the main characteristics for the bench-mark programs
 >>> at the beginning would
 >>> be useful. In particular, the meaning of the 'parameter' in each
 >>> of the benchmarks is
 >>> different (Fig 6 and 8: number of iterations; Fig 7: number of nested
 >>> handlers), and
 >>> this should be highlighted. Possibly, restructure into sub-section,
 >>> to measurement
 >>> the impact of different kinds of params.

Yes, we have added more explanation.

 >>> In general, good analysis of the reasons for performance gains.
 >>> It would help to back up
 >>> the statements by supportive data (isolating the costs for the main
 >>> impact factor), but
 >>> given the focus of the paper I don't think this is strictly necessary.

Yes, we have revised the section within the page limitation.

 >>> What is the same-fringe problem? Give an intuition. And thanks for
 >>> giving a link to the repo!

We are sorry, now you see a short explanation for 'fringe'.

 >>> Reference to "James and Sabry" is missing
```

Fixed.

 >>> Overall: several spelling mistakes in the paper

We have tried our best.

Thank you for all your comments!

$$\boxed{\langle C;\, E;\, K\rangle \longrightarrow_{\text{eff}} \langle C';\, E';\, K'\rangle}$$

$$\langle x;\, E;\, K\rangle \longrightarrow_{\text{eff}} \langle E\,(x)\,;\, E;\, K\rangle \qquad (\textsc{Lookup})$$

$$\langle \texttt{let}\ x = e\ \texttt{in}\ e';\, E;\, K\rangle \longrightarrow_{\text{eff}} \langle e;\, E;\, (\texttt{let}\ x = \Box\ \texttt{in}\ e', E) :: K\rangle \ (\textsc{PushLet})$$

$$\langle w;\, E;\, (\texttt{let}\ x = \Box\ \texttt{in}\ e, E') :: K\rangle \longrightarrow_{\text{eff}} \langle e;\, (x = w) :: E';\, K\rangle \qquad (\textsc{Bind})$$

$$\langle \lambda x.\ e;\, E;\, K\rangle \longrightarrow_{\text{eff}} \langle \texttt{clos}\,(\lambda x.e, E)\,;\, E;\, K\rangle \qquad (\textsc{Close})$$

$$\langle e\ e';\, E;\, K\rangle \longrightarrow_{\text{eff}} \langle e;\, E;\, (\Box\ e', E) :: K\rangle \qquad (\textsc{PushApp})$$

$$\langle w;\, E;\, (\Box\ e, E') :: K\rangle \longrightarrow_{\text{eff}} \langle e;\, E';\, (w\ \Box) :: K\rangle \qquad (\textsc{PushArg})$$

$$\left\langle w;\, E;\, \left(\texttt{clos}\,(\lambda x.e, E)'\ \Box\right) :: K\right\rangle \longrightarrow_{\text{eff}} \langle e;\, (x = w) :: E';\, K\rangle \qquad (\textsc{App})$$

$$\langle \texttt{with}\ h\ \texttt{handle}\ e;\, E;\, K\rangle \longrightarrow_{\text{eff}} \langle h;\, E;\, (\texttt{with}\ \Box\ \texttt{handle}\ e, E) :: K\rangle$$
$$(\textsc{PushWithHandle})$$

$$\begin{array}{c} \langle h;\, E;\, K\rangle \longrightarrow_{\text{eff}} \langle \texttt{closh}\,(h, E)\,;\, E;\, K\rangle \\ \text{where } h = \texttt{handler}\ \mathit{eff}\ \ (\texttt{val}\ \ x \to e_v)\ \ ((x,k) \to e_{\mathit{eff}}) \end{array} \ (\textsc{CloseHandler})$$

$$\left\langle \begin{array}{c} w_h; \\ E'; \\ (\texttt{with}\ \Box\ \texttt{handle}\ e, E) :: K \end{array} \right\rangle \longrightarrow_{\text{eff}} \left\langle \begin{array}{c} e; \\ E; \\ \left((\texttt{with}\ w_h\ \texttt{handle}\ \Box)^{\mathit{eff}}\right) :: K \end{array} \right\rangle$$
$$\text{where } w_h = \texttt{closh}\,(\texttt{handler}\ \mathit{eff}\ (\texttt{val}\ x \to e_v)\ \ ((x,k) \to e_{\mathit{eff}})\,, E)$$
$$(\textsc{Handle})$$

$$\langle \texttt{perform}\ \mathit{eff}\ v;\, E;\, K\rangle \longrightarrow_{\text{eff}} \langle v;\, E;\, (\texttt{perform}\ \mathit{eff}\ \Box) :: K\rangle \ (\textsc{PushPerform})$$

$$\frac{\begin{array}{c} \mathit{split}\,(K, \mathit{eff}\ ) = \left(K', (\texttt{with}\ w_h\ \texttt{handle}\ \Box)^{\mathit{eff}}, K''\right) \\ \text{where } w_h = \texttt{closh}\,(\texttt{handler}\ \mathit{eff}\ \ (\texttt{val}\ x \to e_v)\ \ ((y,k) \to e_{\mathit{eff}})\,, E') \end{array}}{\langle w;\, E;\, (\texttt{perform}\ \mathit{eff}\ \Box) :: K\rangle \longrightarrow_{\text{eff}} \left\langle \begin{array}{c} e_{\mathit{eff}}; \\ (y = w) :: (k = K' * E) :: E'; \\ (\texttt{with}\ w_h\ \texttt{handle}\ \Box)^{\mathit{eff}} :: K'' \end{array} \right\rangle}$$
$$(\textsc{HandlePerform})$$

$$\frac{\begin{array}{c} F = (\texttt{with}\ w_h\ \texttt{handle}\ \Box)^{\mathit{eff}} \\ \text{where } w_h = \texttt{closh}\,(\texttt{handler}\ \mathit{eff}\ \ (\texttt{val}\ x \to e_v)\ \ ((y,k) \to e_{\mathit{eff}})\,, E') \end{array}}{\langle w;\, E;\, F :: K\rangle \longrightarrow_{\text{eff}} \langle e_v;\, (x = w) :: E';\, K\rangle}$$
$$(\textsc{HandleValue})$$

Fig. 12: Semantics of λ_{eff}

$$
\begin{aligned}
FV_p\left(K\ \overrightarrow{pat}\right) &= \bigcup p \in \overrightarrow{pat}.FV_p\left(p\right)\\
FV_p\left(x\right) &= \left\{x\right\}
\end{aligned}
$$

$$
matchable\left(K\ \overrightarrow{v}, K'\ \overrightarrow{pat}\right) = K =_K K' \wedge \forall v \in \overrightarrow{v}, p \in \overrightarrow{pat}.matchable\left(v, p\right)
$$

$$
\theta_1 \oplus \theta_2 = \emptyset \begin{bmatrix} \forall x \in dom\left(\theta_1\right).x \leftarrow \theta_1\left(x\right), \\ \forall y \in dom\left(\theta_2\right).y \leftarrow \theta_2\left(y\right) \end{bmatrix}
$$

$$
\begin{aligned}
genstore\left(K\ \overrightarrow{v}, K\ \overrightarrow{pat}\right) &= \bigoplus_{v \in \overrightarrow{v}, p \in \overrightarrow{pat}} genstore\left(v, p\right)\\
genstore\left(v, x\right) &= \emptyset\left[x \leftarrow v\right]
\end{aligned}
$$

Fig. 13: Auxiliary functions for the semantics of λ_{ac}

$$\langle C[x], \theta\rangle \longrightarrow_{\mathrm{ac}} \langle C[\theta(x)], \theta\rangle \qquad (\textsc{Lookup})$$

$$\frac{x \notin dom\,(\theta)}{\langle C\,[(\lambda x.e)\,v]\,, \theta\rangle \longrightarrow_{\mathrm{ac}} \langle C\,[e]\,, \theta\,[x \leftarrow v]\rangle} \qquad (\textsc{App})$$

$$\frac{x \notin dom\,(\theta)}{\langle C[\texttt{let}\ x = v\ \texttt{in}\ e'], \theta\rangle \longrightarrow_{\mathrm{ac}} \langle C[e], \theta[x \leftarrow v]\rangle} \qquad (\textsc{Let})$$

$$\frac{\forall z \in \left\{f, \overrightarrow{x}, \overrightarrow{g, \overrightarrow{y}}\right\}.z \notin dom\,(\theta)}{\left\langle C\begin{bmatrix}\texttt{let rec}\ f\ \overrightarrow{x} = e_f \\ \overrightarrow{\texttt{and}\ g\ \overrightarrow{y} = e_g} \\ \texttt{in}\ e\end{bmatrix}, \theta\right\rangle \longrightarrow_{\mathrm{ac}} \left\langle C[e], \theta\begin{bmatrix}f \leftarrow \lambda\overrightarrow{x}.e_f, \\ \overrightarrow{g \leftarrow \lambda\overrightarrow{y}.e_g}\end{bmatrix}\right\rangle} \qquad (\textsc{LetRec})$$

$$\frac{l \notin dom\,(\theta)}{\langle C\,[\texttt{create}\ v]\,, \theta\rangle \longrightarrow_{\mathrm{ac}} \langle C\,[l]\,, \theta\,[l \leftarrow v]\rangle} \qquad (\textsc{Create})$$

$$\langle C\,[\texttt{resume}\ l\ v]\,, \theta\rangle \longrightarrow_{\mathrm{ac}} \langle C\,[l : \theta\,(l)\ \ v]\,, \theta\,[l \leftarrow \texttt{nil}]\rangle \qquad (\textsc{Resume})$$

$$\frac{C_2 \text{ does not contains labelled expressions}}{\langle C_1\,[l : C_2\,[\texttt{yield}\ \ v]]\,, \theta\rangle \longrightarrow_{\mathrm{ac}} \langle C_1\,[v]\,, \theta\,[l \leftarrow \lambda x.C_2\,[x]]\rangle} \qquad (\textsc{Yield})$$

$$\langle C\,[l : v]\,, \theta\rangle \longrightarrow_{\mathrm{ac}} \langle C\,[v]\,, \theta\rangle \qquad (\textsc{LabelledReturn})$$

$$\frac{\mathit{eff} =_{\mathit{eff}} \mathit{eff}'}{\langle C\,[\mathit{eff} = \mathit{eff}']\,, \theta\rangle \longrightarrow_{\mathrm{ac}} \langle C\,[\mathit{True}]\,, \theta\rangle} \qquad (\textsc{EqT})$$

$$\frac{\mathit{eff} \neq_{\mathit{eff}} \mathit{eff}'}{\langle C\,[\mathit{eff} = \mathit{eff}']\,, \theta\rangle \longrightarrow_{\mathrm{ac}} \langle C\,[\mathit{False}]\,, \theta\rangle} \qquad (\textsc{EqF})$$

$$\frac{\neg matchable\,(K\ \overrightarrow{v}, pat)}{\left\langle C\begin{bmatrix}\texttt{match}\ K\ \overrightarrow{v}\ \texttt{with} \\ pat\ \ [cond] \rightarrow e; \\ cases\end{bmatrix}, \theta\right\rangle \longrightarrow_{\mathrm{ac}} \langle C\,[\texttt{match}\ K\ \overrightarrow{v}\ \texttt{with}\ cases]\,, \theta\rangle} \qquad (\textsc{MatchNext})$$

$$\frac{\forall x \in FV_p\,(pat)\,.x \notin \mathrm{dom}\,(\theta) \qquad matchable\,(K\ \overrightarrow{v}, pat) \qquad \theta' = \theta \oplus genstore\,(K\ \overrightarrow{v}, pat)}{\langle C\,[\texttt{match}\ K\ \overrightarrow{v}\ \texttt{with}\ pat \rightarrow e; cases]\,, \theta\rangle \longrightarrow_{\mathrm{ac}} \langle C[e], \theta'\rangle} \qquad (\textsc{Match})$$

$$\frac{\forall x \in FV_p\,(pat)\,.x \notin \mathrm{dom}\,(\theta) \qquad matchable\,(K\ \overrightarrow{v}, pat) \qquad \theta' = \theta \oplus genstore\,(K\ \overrightarrow{v}, pat)}{\left\langle C\begin{bmatrix}\texttt{match}\ K\ \overrightarrow{v}\ \texttt{with} \\ pat\ \texttt{when}\ c \rightarrow e; \\ cases\end{bmatrix}, \theta\right\rangle \longrightarrow_{\mathrm{ac}} \left\langle C\begin{bmatrix}\texttt{match}\ c\ \texttt{with} \\ True \rightarrow e; \\ False \rightarrow \\ \quad \texttt{match}\ K\ \overrightarrow{v}\ \texttt{with} \\ \quad cases\end{bmatrix}, \theta'\right\rangle} \qquad (\textsc{MatchWhen})$$

Fig. 14: Semantics of λ_{ac}