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. 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. We have implemented our 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 · One Shot

1 Introduction

Algebraic effects [13] and handlers [14] are an emerging new feature to model effectful computations. They are gaining more and more attention not only from researchers but also from programmers. Algebraic effects and handlers are implemented as a feature in programming languages such as Eff [1], Multicore OCaml [5], and Koka [10], as well as as a library in Haskell [7] [8], OCaml [9], and Scala [2] [4] [5], even in JVM bytecode [3] and C [11], and so on.

Algebraic effects and handlers are implemented in several ways based on stack manipulation, delimited control-operators, or free monad. Unfortunately, all of them have shortcomings as follows. Implementations based on stack manipulation are used in JVM bytecode and C implementations, however, it highly depends on the runtime which needs deep insight on language processors. Therefore implementation cost is rather high, which prevents the feature from being implemented in various systems. Implementation via delimited control-operators is used in OCaml implementation and one of Scala implementations. Although it is a systematic and elegant way to implement algebraic effects and handlers using delimited control-operators since it does not need knowledge on low-level features, unfortunately, few languages have delimited continuation as

[4] https://github.com/atnos-org/eff
[5] https://github.com/typelevel/cats-effect

built-in, and implementing delimited control-operators in a language is non-trivial. Free monad is another systematic way to implement computational effects, and Haskell and one of Scala implementations use it. With free-monad implementation, we only have to concentrate on data structures, realizing the separation of concerns. The demerits of using free monads are that a programmer has to adopt monadic-style, also, to combine other effectful computations with algebraic effects and handlers, we must use monad transformers and individual control operators such as `forM` or `replicateM`, which is messy.

In this paper, we present a new embedding of algebraic effects and handlers that is general, widely available, and easy to understand, compared to other existing ways of implementing them. The key of our embedding is to use coroutines which many languages already have as a built-in feature. Coroutines are less expressive than general delimited-control operators as the former cannot use (implicit) delimited continuations more than once, while the latter can use delimited continuations multiple times. 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 [12]. It is already known that one-shot delimited-control operators can be implemented more efficiently than general, multi-shot ones, thanks to that fact that no copying of continuations is necessary [4]. Similarly, our target feature of algebraic effect and handlers must use continuations at most once. 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. A programmer does not have to write delimited-control operators from effect invocations explicitly because coroutines retrieve the continuation as the rest of a coroutine thread.

We have implemented one-shot algebraic effects and handlers in terms of asymmetric coroutines. Our implementation is practical, as we have made algebraic-effect libraries for Lua [6] and Ruby[7] based on this paper, both of which are already made public. Our libraries have been used by several users, and interested users have ported our libraries to other languages[8][9]. 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. Our libraries allow a user to access continuations, and each library is kept so simple that anyone can understand the implementation.

Our main contributions in this paper are the following.

[6] https://github.com/nymphium/eff.lua
[7] https://github.com/nymphium/ruff
[8] https://github.com/MakeNowJust/eff.js
[9] https://github.com/pandaman64/effective-rust
[10] https://github.com/dry-rb/dry-effects
[11] https://github.com/digital-fabric/affect
[12] https://github.com/briancavalier/forgefx

- We show a new embedding of one-shot algebraic effects and handlers. We use standard asymmetric coroutines only, and no special control features are needed. Hence our embedding applies to various other languages.
- Our embedding does not force programmers to think about the internal data structures. The implementation details are kept transparent.
- 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 paper is organized as follows. Section 2 shows typical examples using algebraic effects and handlers and demonstrates our algebraic-effect library for Lua. Section 3 describes the embedding method by defining the conversion from λ_{eff}, a language with algebraic effect handlers, to λ_{ac}, a language with asymmetric coroutines. 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 with free monads. Section 6 describes related work, and Section 7 concludes.

2 Effectful Programming with *eff.lua*

This section illustrates programming with algebraic effects and handlers by several examples. To express examples, we will use the programming language Lua with our library *eff.lua* to write effectful programs. The library is based on our embedding described in Section 3, which we detail in Section 4.

In the following examples, three functions `inst`, `perform`, and `handler` are used to express algebraic effect and handlers.

2.1 Exception

In our view, algebraic effects are regarded as resumable exceptions, as shown by the following example.

The function `inst` in our library *eff.lua* creates a new effect.

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

We can invoke an effect by calling `perform`.

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

`div` divides `x` by `y` except when `y` is `0`. If `y` is `0`, it performs the effect `DivideByZero`, and the corresponding handler catches the effect. By performing an effect, the control is brought to the nearest handler, similarly to exception handling.

Our library provides the function `handler` to create a new handler.

```
local with_nil = handler {
  val = function(_) return nil end,
  [DivideByZero] = function(_, _)
    return nil
  end
}
```

A handler captures an effect with an argument and the current (delimited) continuation from the effect invocation up to the handler. On the second line in the code above, `val` is a value handler which is used when the handled computation returns the result value. The handler function for `DivideByZero` has two arguments, the first of which receives the argument of the effect invocation and the second is a delimited continuation. In the example, the arguments are ignored, and the whole computation returns `nil`. The code above expresses a simple exception.

As the next step, we show *resumable* exceptions. We implement the handler that catches the effet `DivideByZero` with its continuation and resumes the computation from the point of "throwing an exception."

```
local with_default_zero = handler {
  val = function(v) return v end,
  [DivideByZero] = function(_, k)
    if DEBUG then
      return k(0)
    else
      return nil
    end
  end
}
```

The implementation itself is simple, by simply passing `0` to the continuation. We can use the handler `with_default_zero` as:

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

In this code, we pass `3` and `0` to `div`, and then the effect `DivideByZero` is performed. `with_default_zero` catches the effect and captures the continuation `local v = □; return v + 20` with still handled by `with_default_zero`. The effect handler passes `0` to its continuation, and the computation resumes from

the effect invocation. So `div` returns `0`, and the entire computation results in `0 + 20 = 20` with the value handler, which is an identity function.

2.2 State

We can implement state by a parameter-passing handler without using reference cells.

First, we define two effect operations:

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

To run stateful computations, we define the function `run`, which requires the initial state and computation as a thunk:

```
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
```

`step` is a handler that catches `Get` and `Put`. In order to follow the parameter-passing technuque, the value handler returns a function which itself returns nothing. When the handler catches the effect `Get`, it returns the function that requires a state. When the handler catches the effect `Put` with an actual parameter `v`, it binds `v` to `s` and returns a thunk which passes `s` to the continuation `k` as a new state. After defining the handler, the above code runs the computation with the initial state `init`.

2.3 Deferred functions

We can implement a simple deferred function like Go's `defer`[13]. Go allows a function call on a deferred statement, and then its execution is deferred until the end of the execution of its parent[14]. From the tutorial [15], it is written as:

```
package main

import "fmt"

func main() {
  defer fmt.Println("world")

  fmt.Println("hello")
}
```

In the function `main`, we register the function call `fmt.Println("world")`. After calling `main` and printing `hello`, the deferred function runs and prints `world`.

We can implement it using our library and here show part of it. Go can execute deferred functions even if an exception occurs, but we suppress it for simplicity.

We define the effect `Defer` as follows:

```
local Defer = inst()
```

and the corresponding handler:

```
local with_defer = handler {
  val = function(_) end,
  [Defer] = function(proc, k)
    k()
    proc()
    return nil
  end
}
```

`with_defer` catches the effect `Defer` with the argument `proc`. Since Lua is a call-by-value language and does not have a feature to defer computations, a deferred function is represented as a thunk. When the handler catches the effect `Defer` for the first time, it runs the continuation. After running the continuation, it wastes the result of the continuation and runs `proc` and returns `nil`.

We can use `with_defer` as:

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

[14] Although the syntax accepts any expressions, the de facto standard compiler `go` rejects all expressions but function calls.

[15] https://tour.golang.org/flowcontrol/12

```
with_defer(function()
  perform(Defer, function() print("world") end)

  print("hello")
end)
```

It returns `nil` with printing `hello` and `world`.

To implement the full functionality of `defer` in Go, one must write exception handlers around the continuation and rethrow a caught exception after the deferred function.

3 Embedding Algebraic Effects with Coroutines

This section explains our embedding to implement one-shot algebraic effects and handlers using asymmetric coroutines. For this purpose, we define λ_{eff}, a language which has *one-shot* algebraic effects, λ_{ac}, a language which has asymmetric coroutines, and then a program translation from λ_{eff} to λ_{ac}.

3.1 λ_{eff}

λ_{eff} is an untyped language with algebraic effects and handlers based on Effy [15].

Syntax Figure 1 defines the syntax of λ_{eff}.

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

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

The expression `perform` *eff* v invokes an effect *eff* with an argument v. A usual `let` binding is written as `let` $x = c_1$ `in` c_2.

The expression `handler` *eff* $(\texttt{val}\ x \to e_1)\,((y, k) \to e_2)$ creates a handler, which catches the effect *eff* and returns 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 performed (invoked). The expression `val` $x \to e_1$ gives a value handler, namely, a handler which is used when the body of a handler returns normally.

For simplicity, λ_{eff} can handle only one effect per handler, whereas handlers in Effy can cope with multiple effects. Although this simplification restricts the usability of our language, we can easily extend it to allow multiple effect operations per handler. In fact, the actual implementation of our library provides multiple handlers stated in Section 4.

The expression `with` h `handle` e is a handling expression, which sets the handler h and then evaluates e under the handler.

The syntactic category w is a runtime value whereof $\texttt{clos}\,(\lambda x.e, E)$ and $\texttt{closh}\,(h, E)$. $\texttt{clos}\,(\lambda x.e, E)$ is a λ-term with a closing environment E, and $\texttt{closh}\,(h, E)$ is a handler with E.

The frame F is either an application, the `let` binding, or handling expressions, and C represents the "code" component in the CEK machine, and K stands for a continuation, which consists of a stack of frames.

Semantics

Helper functions We introduce three helper functions for semantics in Figure 2: $K \mathbin{/\!/} \mathit{eff}$ 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. If more than one frames handle

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

Fig. 2: Helper Functions for Semantics

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.

Small-step semantics Figure 3 defines the small-step, call-by-value, left-to-right semantics ($\longrightarrow_{\text{eff}}$) in the CEK-machine style. 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 more or less 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* $\square$ 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 $(\texttt{with}\ w_h\ \texttt{handle}\ \square)^{\mathit{eff}}$ 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.

3.2 λ_{ac}

The language λ_{ac} is based on one of the calculi defined by de Moura and Ierusalimschy [12], namely, it is based on the calculus for stackful asymmetric coroutines. For practical reasons, we added to this language `let` with recursion, pattern matching, and comparison operators.

Syntax Figure 4 defines the syntax of λ_{ac}. We will use λ_{ac} as the target language of program transformation from λ_{eff}, so it has several program constructs for this purpose. *Effects* and each element correspond to the effects in λ_{eff}.

We added conditional expression, pattern matching, and (mutual) recursion to de Moura and Ierusalimschy's calculi. $f\ \overrightarrow{x}$ is an abbreviation of $f\ x_0\ x_1\ \cdots\cdots\ x_n$ and $\overrightarrow{\texttt{and}\ g\ \overrightarrow{y} = e}$ is of $\texttt{and}\ g_0\ \overrightarrow{y} = e_0\ \texttt{and}\ g_1\ \overrightarrow{y} = e_1\ \texttt{and}\ \cdots\cdots \texttt{and}\ g_m\ \overrightarrow{y} = e_m$. A similar abbreviation is applied to constructors and pattern matching.

Constructs for asymmetric coroutines are labels, labelled expression $l : e$, `create`, `resume`, and `yield`. A label is a reference to a coroutine, and a labelled expression $l : e$ is an expression e in the coroutine l.

K represents constructors; for instance, *True* and *False* are boolean constants. 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 3: Semantics of λ_{eff}

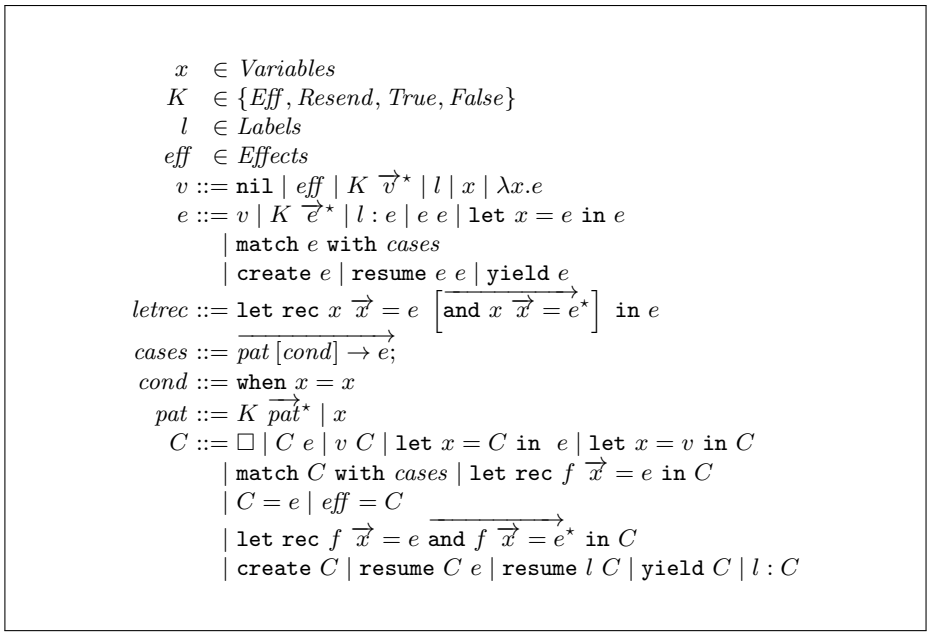


Fig. 4: the syntax of λ_{ac}

We define the semantics of the language λ_{ac} in the same way as de Moura and Ierusalimschy. Due to lack of space we state it in Section A in the appendix of this paper.

3.3 Translating λ_{eff} to λ_{ac}

In this section, we present a program translation from λ_{eff} to λ_{ac}. Our translation is syntax-directed and compositional. Figure 5 defines the translation. For a λ_{eff}-term e, its translation $[\![e]\!]\eta$ is a λ_{ac}-term where η is a finite map from λ_{eff}-variables to λ_{ac}-values. The map η is used to translate free variables in e. We assume that η is extended by the syntax $\eta\,[x \to x']$, which maps x to x', and y to $\eta\,(y)$ for any y in the domain of η.

The translation is homomorphic for several expressions, including variable dereference, λ-abstraction, applications, and `let`. We map effects $[\![\mathit{eff}]\!]$ to the same effect, based on the assumption that the source and target languages share effects (more precisely, effect names).

The translation maps `perform` to `yield` by the following observation. In algebraic effects, when an effect invocation occurs, the control is transferred to a handler corresponding to an effect, while, in coroutines, when a `yield` is called

in a coroutine, the control is transferred to `resume`, which resumes the 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.

The handling expression `with` h `handle` e is translated to a simple application because the handler is mapped to a function.

The translation for a handler (the last case) is highly non-trivial. We first define a function *handler* in λ_{ac} and pass the translated semantic objects to it.

The recursive function *handler* takes four arguments: *eff* as an effect, *vh* as a value handler, *effh* as an effect handler, and *th* as a thunk of computation to be handled. It creates a coroutine *co* by the thunk *th*, and also three functions *continue*, *rehandle*, and *handle*.

The function *continue* requires an argument and passes it to `resume` *co*, so it runs the rest of the coroutine *co*, or we can say, *continue* runs the *continuation*. Then *continue* passes the result of the continuation to *handle*. Handling the result of the continuation makes the handler to be a *deep* handler.

The function *rehandle* creates a new handler with handling k arg. It uses *continue* as a value handler, so the expression finally continues the computation of *co*.

The function *handle* receives r as an argument, which is a yielded value by the handled expression. It pattern-matches the tag of r and dispatches the control. The first arm of the pattern match (Eff eff' v `when` $eff' = eff$) is to handle an effect invocation. During the computation of the handled expression, if an effect *eff'* is performed with the argument v, then r has the form Eff eff' v. If eff' equals to *eff*, it is the effect this handler can handle, and it passes v and *continues* to the effect handler *effh*. The second arm is the case when this handler cannot handle the effect, in which case the function yields $Resend$ r *continue* to re-invoke the effect. The third and fourth arms are to resolve effects thrown from the second arm above. The third is the case the handler can handle when eff' equals to *eff*. It applies *effh* such as the first arm, except passing *rehandle* k as a continuation. k is the continuation of the inner handler and *continue* is the continuation of the expression the current handler handles. In order to run the later continuation after the former continuation is executed, we wrap k with *rehandle*, which contains the "current" continuation.

rehandle also adjusts the layer of coroutines. In the second arm, *handle* calls `yield`, so the control runs away from one coroutine. We could write $\lambda arg.\ handle\ (k\ arg)$ instead of *rehandle* k, only to treat the return value of the continuation. If we did it, the layer of coroutines would decrease, and finally, we would get an error calling `yield` outside of coroutine. So *rehandle* encapsulates the expression with coroutine internally and avoid to decreasing the layer of coroutines.

The fourth arm rethrows the effect which the handler cannot handle. This case is almost the same as the second arm, but it modifies the continuation for the same reason described in the third arm. The last arm accepts any other value and passes the value to the value handler. After defining functions, *handler* runs *continue* with the argument `nil`.

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

where $\mathit{handler} =$

```
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 (λ_.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. 5: Translation from λ_{eff} to λ_{ac}

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. For this simplicity, we note that a few people have already ported our translation to other languages.

4 Implementation

We have implemented algebraic effects and handlers in Lua and Ruby based on the translation described in Section 3. Since the translation is local and compositional, and the variables can be managed on the host language, we can realize our implementation as a library.

The implementations are simple and easy to understand, however, several issues have arisen in the process of implementation we will address below.

Multiple Effect Handler In this paper, we put the restriction that a handler may catch only one effect in λ_{eff}. However, this restriction is only for the presentation purpose. In fact, we have eliminated this restriction in our actual implementation so that one handler may catch multiple effects and all examples (including the examples in this paper) that use multiple effects per handler run without problems. 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 instances. Again this is due to simplicity, and we can eliminate this restriction in our actual implementation. The merit of allowing dynamic creation of effect instances is that a certain kind of effectful programs does need the uniqueness of effect instances, for instance, Kiselyov's examples[9].

Conflict with Other Effects An assumption on our translation is that all effects are written via algebraic effects and handlers. If our source program uses other effects besides algebraic effects and handlers, 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 coroutine with algebraic effects, 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) algebraic effects and handlers. Representing coroutines and other effects using algebraic effects and handlers is possible, but tedious if the language has coroutines from the beginning.

This problem can be solved as follows, thanks to the expressivity of algebraic effects and handlers. 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,
      [Yield] = function(u, k)
        co.it = k
        return u
      end
    })(function()
      return co.it(v)
    end)
  end
end
```

The code in the last half is an implementation of coroutines by algebraic effects 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 5. The function `create` 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 of `resume` catches the invocation of `yield` with an argument and a continuation. This continuation is the rest of computation of the coroutine, so the handler stores the continuation to the cell and returns the value `u`. 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. 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 know several solutions to this direction, clearly we need to do more to combine different kinds of effects in a single program.

5 Evaluation

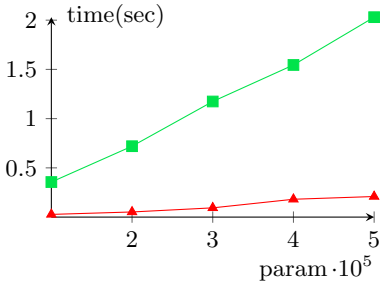
We have conducted experiments on microbenchmark using our library in Lua, and implementation in Lua based on free monads [15], and compare their performance. All the code for the benchmark is publicly available in the GitHub repository[16]. In the following figures, the symbol ▲ represents the result of ours 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 ⋆ 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

[16] 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

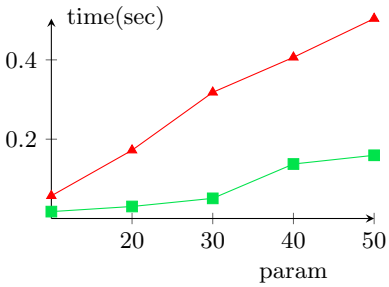


param ($10^5\times$)	1	2	3	4	5
free	0.3560	0.7199	1.1741	1.5449	2.0303
ours	0.0273	0.0521	0.0939	0.1811	0.2090

Fig. 6: Result of `onestate` benchmark

benchmark uses the function `count`, cited from [7], 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 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

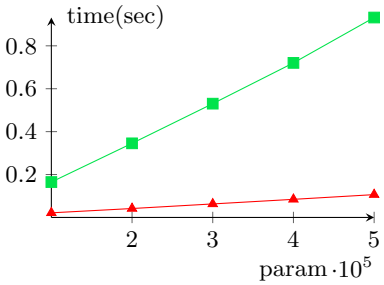


param ($10\times$)	1	2	3	4	5
free	0.0174	0.0303	0.0508	0.1374	0.1594
ours	0.0572	0.1722	0.3181	0.4064	0.5041

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 5, 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

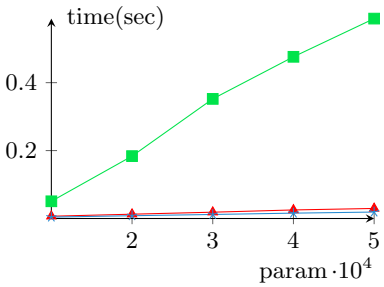


param ($10^5\times$)	1	2	3	4	5
free	0.1650	0.3455	0.5304	0.7204	0.9319
ours	0.0215	0.0416	0.0630	0.0844	0.1064

Fig. 8: Result of `looper` benchmark

`for`-loop and set a handler out of the loop to catch the effect. Our library runs 9 times as fast as the free-monad based implementation. It should also be noted that free monads require the `forM`-operator rather than the `for`-operator, which has a particular overhead. Again an advanced compiler may be able to eliminate all or part of this kind of monadic overhead.

Figure 9 shows the result of the benchmark, which solves the same-fringe problem by using algebraic effects and coroutines. The benchmark creates a tree,



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

runs a solution for the same fringe problem, and measures the performance. The benchmark gives 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 algebraic effects and handlers is advantageous in several programming languages from the performance viewpoint. We also want to emphasize that writing effectful programs using coroutines is harder than writing the same programs using algebraic effects and handlers, 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* [6], and it has its own merits. We have also implemented the shallow handler with coroutines. The idea is simple; after the handler catches the effect, the handler always resends effects to the outer handler. We have explained the role of *rehandle* in Figure 5 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. 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. 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[5]. They provide one-shot continuations due to the performance problem, and if multi-shot continuations are needed, they allow explicit copy for continuations.

React, a popular web framework for JavaScript has a utility called Hooks[17], which makes components with side-effects modular. Abramov, who is one of React developers, stated the relevance between Hooks and algebraic effects in

[17] https://reactjs.org/docs/hooks-reference.html

his blog post[18]. We think we can simulate the abilities of Hooks by one-shot algebraic effects.

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, liearly used (delimited) continuations, or linear continuation-passsing 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.

References

1. Bauer, A., Pretnar, M.: Programming with Algebraic Effects and Handlers. Journal of Logical and Algebraic Methods in Programming **84**, (03 2012). https://doi.org/10.1016/j.jlamp.2014.02.001
2. Brachthäuser, J., Schuster, P.: Effekt: extensible algebraic effects in Scala (short paper). pp. 67–72 (10 2017). https://doi.org/10.1145/3136000.3136007
3. Brachthäuser, J., Schuster, P., Ostermann, K.: Effect handlers for the masses. Proceedings of the ACM on Programming Languages **2**, 1–27 (10 2018). https://doi.org/10.1145/3276481
4. Bruggeman, C., Waddell, O., Dybvig, R.: Representing Control in the Presence of One-Shot Continuations. vol. 31, p. (02 1970). https://doi.org/10.1145/249069.231395
5. Dolan, S., White, L., Madhavapeddy, A.: Multicore OCaml. In: OCaml Users and Developers Workshop (2014)
6. Hillerström, D., Lindley, S.: Shallow Effect Handlers. In: Asian Symposium on Programming Languages and Systems. pp. 415–435. Springer (2018)
7. Kammar, O., Lindley, S., Oury, N.: Handlers in Action. vol. 48, pp. 145–158 (09 2013). https://doi.org/10.1145/2500365.2500590

[18] https://overreacted.io/algebraic-effects-for-the-rest-of-us/

8. Kiselyov, O., Ishii, H.: Freer Monads, More Extensible Effects. ACM SIGPLAN Notices **50**, (03 2015). https://doi.org/10.1145/2887747.2804319
9. Kiselyov, O., Sivaramakrishnan, K.: Eff Directly in OCaml. Electronic Proceedings in Theoretical Computer Science **285**, 23–58 (12 2018). https://doi.org/10.4204/EPTCS.285.2
10. Leijen, D.: Algebraic Effects for Functional Programming. Tech. rep., Technical Report. 15 pages. (2016)
11. Leijen, D.: Implementing Algebraic Effects in C. pp. 339–363 (11 2017). https://doi.org/10.1007/978-3-319-71237-6_17
12. Moura, A., Ierusalimschy, R.: Revisiting Coroutines. ACM Transactions on Programming Languages and Systems **31**, (07 2004). https://doi.org/10.1145/1462166.1462167
13. Plotkin, G., Power, J.: Algebraic Operations and Generic Effects. Applied Categorical Structures **11**, 69–94 (02 2003). https://doi.org/10.1023/A:1023064908962
14. Plotkin, G., Pretnar, M.: Handling Algebraic Effects. Logical Methods in Computer Science **9**, (12 2013). https://doi.org/10.2168/LMCS-9(4:23)2013
15. 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 $\boldsymbol{\lambda_{ac}}$

Auxiliary functions Figure 10 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

$$FV_p\left(K\ \overrightarrow{pat}\right) = \bigcup p \in \overrightarrow{pat}.FV_p\,(p)$$
$$FV_p\,(x) = \{x\}$$

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

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

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

Fig. 10: Auxiliary functions for the semantics of λ_{ac}

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 11 shows the operational semantics of λ_{ac} by the transition ($\longrightarrow_{\mathrm{ac}}$) 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 $\theta\,(l) = \texttt{nil}$, 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 associatedvalue 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. 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.

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

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

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

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

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

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

$$\langle C_1\,[l : C_2\,[\texttt{yield}\ \ v]]\,, \theta\rangle \longrightarrow_{\mathrm{ac}} \langle C_1\,[v]\,, \theta\,[l \leftarrow \lambda x.C_2\,[x]]\rangle \qquad \text{(Yield)}$$

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

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

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

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

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

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

Fig. 11: Semantics of λ_{ac}