Retrofitting Effect Handlers onto OCaml

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer and Anil Madhavapeddy

Abstract

Effect handlers have been gathering momentum as a mechanism for modular programming with user-defined effects. Effect handlers allow for non-local control flow mechanisms such as generators, async/await, lightweight threads and coroutines to be composably expressed. We present a design and evaluate a full-fledged efficient implementation of effect handlers for OCaml, an industrial-strength multi-paradigm programming language. Our implementation strives to maintain the backwards compatibility and performance profile of existing OCaml code. Retrofitting effect handlers onto OCaml is challenging since OCaml does not currently have any non-local control flow mechanisms other than exceptions. Our implementation of effect handlers for OCaml: (i) imposes negligible overhead on code that does not use effect handlers; (ii) remains compatible with program analysis tools that inspect the stack; and (iii) is efficient for new code that makes use of effect handlers.

1 Introduction

Effect handlers [50] provide a modular foundation for user-defined effects. The key idea is to separate the effectful operations from their interpretations, which are given by handlers of the effects. For example,

```ocaml
let rec gsearch e =
  match e with
  | Val v -> v (* value case *)
  | Div (e1, e2) ->
    match eval e1, eval e2 with
    | Z v1, Z v2 ->
      if v2 = 0 then perform (DivBy0 v1)
      else Z (v1/v2)
    | Error, _ -> Error
    | Inf s, _ -> Inf s
    | ... k
  | DivBy0 _ k ->
    effect (DivBy0 _) k -> continue k Error (* effect case *)
```

Line 4 declares an effect DivBy0 that is parameterised by an integer, which when performed returns the result of division-by-zero. If the denominator of the division is 0, then we perform the DivBy0 effect with the numerator value (line 12) so that the context may decide the return value. We obtain the iOS semantics by wrapping the evaluator with this handler:

```ocaml
let ios e = match eval e with
  | v -> v (* value case *)
  | effect (DivBy0 _) k -> continue k Error (* effect case *)
```

If the evaluator returns a value, we simply return that. If the evaluator performs the DivBy0 effect, we catch the effect in the pattern match along with the continuation k of the corresponding perform delimited by this handler. For iOS, we continue the suspended computation with an Error value and the computation ios(1/0) returns Error. We can get the gsearch semantics by enclosing the evaluator in this handler:

```ocaml
let gsearch e = match eval e with
  | v -> v
  | effect (DivBy0 v) k ->
    if v = 0 then continue k Error
    else if v > 0 then continue k (Inf P)
    else continue k (Inf N)
```

Executing gsearch(1/0) returns positive infinity (Inf P).

1.1 An effectful calculator

Effect handlers provide a structured form of delimited continuations [16] that enables non-local control-flow mechanisms such as resumable exceptions, coroutines, generators and asynchronous I/O to be composably expressed. For example, consider the task of handling division-by-zero in a calculator application. Different calculators handle division-by-zero differently. In the iOS 13 Calculator (ios), division-by-zero returns Error. In the Google search calculator (gsearch) 0/0 returns Error, but for n!=0 the n/0 returns Infinity with the same sign as n. How do we capture all these different behaviours without changing the evaluator?

Common Lisp Conditions [56] are an elegant way to solve this problem. Unlike exceptions where the library decides whether an error is fatal or not, conditions allow the client of the library to make that decision. Conditions provide a restart mechanism for non-fatal errors to resume execution from the point where the condition was signalled. On division-by-zero, the calculator signals the handler and lets the handler decide the course of action based on the desired semantics. We now implement this using effect handlers in OCaml syntax:

```ocaml
let rec eval e =
  match e with
  | Val v -> v
  | ... k
  | Div (e1, e2) ->
    match eval e1, eval e2 with
    | Z v1, Z v2 ->
      if v2 = 0 then perform (DivBy0 v1)
      else Z (v1/v2)
    | Error, _ -> Error
    | Inf s, _ -> Inf s
    | ... k
```

```ocaml
let ios e = match eval e with
  | v -> v (* value case *)
  | effect (DivBy0 _) k -> continue k Error (* effect case *)
```

```
let gsearch e = match eval e with
  | v -> v
  | effect (DivBy0 v) k ->
    if v = 0 then continue k Error
    else if v > 0 then continue k (Inf P)
    else continue k (Inf N)
```

Executing gsearch(1/0) returns positive infinity (Inf P).
1.2 Asynchronous I/O

We can modularly extend the calculator to read user input using the `Read` effect defined earlier as follows:

```ocaml
type exp = ... | Inp

let rec eval e = match e with
| ... |
| Inp -> int_of_string (perform (Read stdin))

The `Read` effect can then be handled by another handler enclosing the division-by-zero handler:

```ocaml
let run e = match gsearch e with
| v -> v
| effect (Read chan) k ->
  register_async_input_line (chan, k);
  continue (get_ui_k ()) ()
```

which registers the read operation to be performed asynchronously with the continuation as the callback. Subsequently, it resumes the user-interface continuation so that the application remains responsive. Observe that neither the division-by-zero handler nor the I/O handler needs to be aware of the other.

1.3 Retrofitting effect handlers

There are many research languages and libraries built around effect handlers [4, 8, 13, 29, 39]. Unlike these efforts, our goal is to retrofit effect handlers onto the OCaml programming language, which has been in continuous use for the past 25 years in large codebases including verification tools [5, 14], mission critical software systems [45] and latency sensitive networked applications [44]. OCaml is particularly favoured for its competitive yet predictable performance, with a fast foreign-function interface (FFI). It has excellent compatibility with program analysis tools such as debuggers and profilers that utilise DWARF stack unwind tables [20] to obtain a backtrack. Our primary motivation for adding effect handlers to OCaml is to support: (i) highly concurrent applications such as web servers using a natural `direct-style` as opposed to using callbacks; and (ii) efficient nested parallel programs in the vein of Cilk [9], OpenMP [33], X10 [11], Intel TBB [32], and variants of Parallel ML [25, 55, 59].

OCaml currently does not support any non-local control flow mechanisms other than exceptions. This makes it particularly challenging to implement the delimited continuations necessary for effect handlers without sacrificing the desirable properties of OCaml. One standard way of implementing continuations is to use continuation-passing style (CPS) in the compiler’s intermediate representation (IR) [39]. OCaml does not use a CPS IR, and changing the compiler to utilise a CPS IR would be an enormous undertaking that would affect the performance profile of existing OCaml applications due to the increased memory allocations as the continuation closures get allocated on the heap [22]. Moreover, with CPS, stack unwinding is meaningless and hence, we would lose compatibility with tools that inspect the program stack. Hence, we choose not to use CPS translation and represent the continuations as call stacks.

The search for an expressive effect system that guarantees that all the effects performed in the program are handled (effect safety) in the presence of advanced features such as polymorphism, modularity and generativity is an active area of research [6, 7, 29, 39]. We do not focus on this question in this paper, and our implementation of effect handlers in OCaml does not guarantee effect safety. For example, running `perform` (Read stdin) without an enclosing handler raises Unhandled exception at the point of `perform`. We leave the question of effect safety for future work.

1.4 Requirements

We motivate our effect handler design based on the following ideal requirements:

**R1: Backwards compatibility.** A well-behaved OCaml program does not break under OCaml extended with effect handlers. OCaml code that does not use effect handlers will pay negligible performance and memory cost.

**R2: Tool compatibility.** OCaml programs with effect handlers produce well-formed backtraces and remain compatible with program analysis tools such as debuggers and profilers that inspect the stack using DWARF unwind tables.

**R3: Effect handler efficiency.** The program must accommodate millions of continuations at the same time to support highly-concurrent applications. Installing effect handlers, capturing and resuming continuations must be fast.

**R4: Forwards compatibility.** While this work does not present an effect system, the design must be compatible with a future effect system for OCaml. As a cornerstone of modularity, we also want blocking I/O code to be made asynchronous with the help of effect handlers.

The need to host millions of continuations at the same time rules out the use of a large contiguous stack space as in C for continuations. Instead, we resort to using small initial stacks and growing the stacks on demand. As a result, OCaml functions, irrespective of whether they use effect handlers, need to perform stack overflow checks, and external C functions (which do not have stack overflow checks) must be performed on a separate system stack. Additionally, we must generate DWARF stack unwind tables for stacks that may be non-contiguous. In this work, we develop the compiler and runtime support for implementing efficient effect handlers for OCaml that satisfy these requirements.

Our work is also timely. The WebAssembly [28] community group is considering effect handlers as one of the mechanisms for supporting concurrency, asynchronous I/O and generators [58]. Project Loom [42] is an OpenJDK project that adds virtual threads and delimited continuations to Java. The Swift roadmap [57] includes direct-style asynchronous programming and structured concurrency as milestones. We believe that our design choices will inform similar choices to be made in other industrial-strength languages.
1.5 Contributions

Our contributions are to present:

- the design and implementation of one-shot effect handlers for OCaml. Our design retains OCaml’s compatibility with program analysis tools that inspect the stack using DWARF unwind tables. We have validated our DWARF unwind tables with the assistance of an automated validator tool [3].
- a formal operational semantics for the effect handler implementation in OCaml.
- extensive evaluation which shows that our implementation has negligible impact on code that does not use effect handlers, and serves as an efficient foundation for scalable concurrent programming.

We have implemented effect handlers in a multicore extension of the OCaml programming language which we call Multicore OCaml to distinguish it from stock OCaml.

The remainder of the paper continues with a description of the stock OCaml program stack (§2). We then refine the design further by focussing on retrofitting effect handlers into a mainstream systems language (§3), followed by the compilation scheme and operational semantics for Multicore OCaml effect handlers (§4). We then discuss the compiler and the runtime system support for implementing effect handlers (§5), and present an extensive performance evaluation of effect handlers (§6) against our design goals (§1.4). Finally, we discuss the related work (§7) and conclude (§8).

2 Background: OCaml Stacks

The main challenge in implementing effect handlers in Multicore OCaml is managing the program stack and preserving its desirable properties. In this section, we provide an overview of the program stack and related mechanisms in stock OCaml.

Consider the layout of the stock OCaml stack for the program shown in Figures 1a and 1b. The OCaml main function `main` installs two exception handlers `h1` and `h2` to handle the exceptions `E1` and `E2`. `main` calls the external C function `ocaml_to_c`, which in turn calls back into the OCaml function `c_to_ocaml`, which raises the exception `E1`. OCaml supports raising exceptions in C as well as throwing exceptions across external calls. Hence, the exception `E1` gets caught in the handler `h1`, and `main` returns 42. The layout of the stack in the native code backend just before raising the exception in `c_to_ocaml` is illustrated in Figure 1c. Note that the stack grows downwards.

OCaml uses the same program stack as C, and hence the stack has alternating sequences of C and OCaml frames. However, unlike C, OCaml does not create pointers into OCaml frames. OCaml uses the hardware support for `call` and `return` instructions for function calls and returns. OCaml does not perform explicit stack overflow checks in code, and, just like C, relies on the guard page at the end of the stack region to detect stack overflow. Stack overflow is detected by a memory fault and a `StackOverflow` exception is raised to unwind the stack.

2.1 External calls and callbacks

OCaml does not use the C calling convention. In particular, there are no callee-saved registers in OCaml. In the x86-64 backend, the OCaml runtime makes use of two C callee-saved registers for supporting OCaml execution. The register `r15` holds the allocation pointer into the minor heap used for bump pointer allocation, and `r14` holds a reference to the `Caml_state`, a table of global variables used by the runtime. This makes external calls extremely fast in OCaml. If the external function does not allocate in the OCaml heap, then it can be called directly and no bookkeeping is necessary. For external functions which allocate in the OCaml heap, the cached allocation pointer is saved to `Caml_state` before the external call and it is restored on return. Similarly, callbacks into OCaml are also cheap: these involve loading the arguments in the right registers and calling the OCaml function. OCaml callbacks are relatively common as the garbage collector (GC) finalisation functions are executed as OCaml callbacks.

2.2 Exception handlers

The lack of callee-saved registers also makes exception handling fast as there are no callee-saved registers to be restored when the exception handler runs. Installing an exception handler simply pushes the program counter (pc) of the handler and the current exception pointer (`exn_ptr`) to the stack. After this, the current exception pointer is updated to be the current stack pointer (`rsp`). This creates a linked-list of exception handler frames on the stack as shown in Figure 1c. Raising an exception simply sets `rsp` to `exn_ptr`, loads the saved `exn_ptr`, and jumps to the pc of the handler.

In order to forward exceptions across C frames, the C stub function `caml_call_ocaml`, which calls the external C function `c_to_ocaml`, which raises the exception `E1`. OCaml supports raising exceptions in C as well as throwing exceptions across external calls. Hence, the exception `E1` gets caught in the handler `h1`, and `main` returns 42. The layout of the stack in the native code backend just before raising the exception in `c_to_ocaml` is illustrated in Figure 1c. Note that the stack grows downwards.

OCaml uses the same program stack as C, and hence the stack has alternating sequences of C and OCaml frames. However, unlike C, OCaml does not create pointers into OCaml frames. OCaml uses the hardware support for `call` and `return` instructions for function calls and returns. OCaml does not perform explicit stack overflow checks in code, and, just like C, relies on the guard page at the end of the stack region to detect stack overflow. Stack overflow is detected by a memory fault and a `StackOverflow` exception is raised to unwind the stack.

2.3 Stack unwinding

OCaml generates `stack maps` in order to accurately identify roots on the stack for assisting the GC. For every call point in the program, the OCaml compiler emits the size of the frame and the set of all live registers in the frame containing pointers into the heap. During a GC, the OCaml stack is walked and the roots are marked, skipping over the C frames. OCaml also generates precise DWARF unwind information for OCaml, thanks to which debuggers such as `gdb` and
11db, and profilers such as perf work out-of-the-box. For example, for the program in Figures 1a and 1b, one could set a break point in gdb at caml_raise_exn to get the backtrace in Figure 1d which corresponds to the stack in Figure 1c.

The same backtrace can also be obtained by using frame pointers instead of DWARF unwind tables. OCaml allows compiling code with frame pointers, but they are not enabled by default. The OCaml stack tends to be deep with small frames due to the pervasive use of recursive functions, not all of which are tail-recursive. Hence, the addition of frame pointers can significantly increase the size of the stack\(^1\). Moreover, not using frame pointers saves two instructions in the function prologue and epilogue, and makes an extra register (rbp on x86_64) available. Note that the DWARF unwind information is complementary to the information used by OCaml to walk the stack for GC.

### 3 Refining the design

In this section, we refine the effect handler design for the OCaml use case. The primary motivation for adding effect handlers to OCaml is to support a non-local control flow mechanism to implement asynchronous I/O in direct style, generators and enable task parallelism. These use cases do not resume the continuations more than once. Hence, our continuations are one-shot, and resuming the continuation more than once raises an Invalid_argument exception. It is well-known that one-shot continuations can be implemented efficiently [10].

While OCaml permits throwing exceptions across C frames, we do not allow effects to propagate across C frames as the C frames would become part of the captured continuation. Managing C frames as part of the continuation is a complex endeavour [38], and we find that the complexity budget outweighs the relatively fewer mechanisms enabled by this addition in our setting.

The interaction of non-local control flow with systems programming is quite subtle [19, 40]. Consider the following function that uses blocking I/O functions from the OCaml standard library to copy data from the input channel \(ic\) to the output channel \(oc\):

```ocaml
let copy ic oc =
  let rec loop () =
    output_string oc ((input_line ic) ^ "\n");
    loop ()
  in
  try loop () with
  | End_of_file -> close_in ic; close_out oc
  | e -> close_in ic; close_out oc; raise e
```

The function `input_line` raises an `End_of_file` exception on reaching the end of input, which is handled by the exception handler which closes the channels. The `close_*` functions do nothing if the channel is already closed. The code is written in a defensive style to handle other exceptional cases such as the channels being closed externally. Both `input_line` and `output_string` raise `Sys_error` exception if the channel is closed. In this case, the catch-all exception handler closes the channels and reraises the exception to communicate the exceptional behaviour to the caller.

---

\(^1\)https://github.com/ocaml/ocaml/issues/5721#issuecomment-472965549
One of our goals (§1.4) is to make this code transparently asynchronous. We can define effects for performing the I/O operations and wrap them up in functions with the same signature as the one from the standard library:

```ocaml
effect In_line : in_channel -> string
effect Out_str : out_channel * string -> unit
let input_line ic = perform (In_line ic)
let output_string oc s = perform (Out_str (oc, s))
```

We can then use a top-level I/O handler, similar to the introductory example (§1.2), to discharge the I/O operations asynchronously and resume with the result. This handles value return cases, but what about the exceptional cases `End_of_file` and `Sys_error`? To this end, we introduce a `discontinue` primitive to resume a continuation by raising an exception. In this example, on reaching the end of file, we would discontinue the captured continuation of the `input_line` function with `discontinue k End_of_file`, which raises the exception at `input_line`, and the open channels will be closed.

Ocaml programs that use resources such as channels are usually written defensively with the assumption that calling a function will return exactly once, either normally or exceptionally. Since effect handlers in Multicore Ocaml do not ensure that all the effects are handled, if the function performs an effect with no matching handler, then the function will not return at all. To remedy this, when such an effect bubbles up to the top-level, we discontinue the continuation with an unhandled exception so that the exception handlers may run and clean up the resources.

## 4 Semantics

In this section, we formalise the effect handler design for Multicore Ocaml.

### 4.1 Static semantics

As mentioned earlier, effect handlers in Multicore Ocaml do not guarantee effect safety, but only guarantee type safety. Programs without matching effect handlers are well-typed Multicore Ocaml programs. As a result, our static semantics is simpler than languages that ensure effect safety [4, 6, 13, 29, 39, 52]. This is important for backwards compatibility as our goal is to retrofit effect handlers to a language with large legacy codebases; programs that do not use effects remain well-typed, and those that do compose well with those that don’t.

The static semantics of effect handlers in Ocaml is captured succinctly by its API:

```ocaml
(type 'a eff = ..
type ('a, 'b) continuation
val perform : 'a eff -> 'a
val continue : ('a, 'b) continuation -> 'a -> 'b
val discontinue : ('a, 'b) continuation -> exn -> 'b
(* Internal API *)
type 'a comp = unit -> 'a
type ('a, 'b) handler =
(retc: 'a -> 'b;
effc: 'c. 'c eff -> ('c, 'b) continuation -> 'b; )
val match_with : 'a comp -> ('a, 'b) handler -> 'b
```

We introduce an extensible variant type [49] `a eff` of effect values, which when performed using the `perform` primitive returns an `a` value. Constructors for the value of type `a eff` are declared using the effect declarations. For example, the declaration `effect E : string -> int is syntactic sugar for adding a new constructor to the variant type `type eff += E : string -> int eff`. We introduce the type `('a, 'b) continuation` of delimited continuations which expects a `a` value for resumption and returns a `b` value. The continuations may be continued with a suitably typed value or discontinued with an exception.

For handling the effects, our implementation extends Ocaml’s

```ocaml
match ... with syntax with effect patterns. The expression

```mismatch e with
| None -> false | Some b -> b
| effect (E s) k1 -> e1 | effect (F f) k2 -> e2
```

is translated to the equivalent of

```ocaml
match_with (fun () -> e)
{ retc = (function None -> false | Some b -> b);
  effc = (function
    | (E s) -> (fun k1 -> e1)
    | (F f) -> (fun k2 -> e2)
  )
  | e -> (fun k -> match perform e with
       | v -> continue k v
       | exception e -> discontinue k e)); }
```

For the sake of exposition, we introduce a `('a, 'b) handler` type. This handler handles a `a comp` that returns a `a` value, and itself returns a `b` value. The handler has a return case `retc` of type `a -> b`. The effect case `effc` handles effects of type `c eff` with `('c, 'b) continuation` and returns a value of type `b`. The last case in `effc` `reperforms` any unmatched effect to the outer handler and returns the value and exceptions back to the original performer. In the implementation, `reperform` is implemented as a primitive to avoid executing code on the resumption path.

### 4.2 Dynamic semantics

We present an operational semantics for a core language of effect handlers that faithfully captures the semantics of the Multicore Ocaml implementation. An executable version of the semantics, implemented as an Ocaml interpreter, along with examples, is included in the supplementary material.

#### 4.2.1 Syntax

Our expressions (Figure 2a) consist of integer constants (`n`), variables (`x`), abstraction (Λx. `x`), application (e e), arithmetic expressions (e ◦ e) where ◦ ranges over (+, −, *, /), raising exceptions (raise `l e`), performing effects (perform `l e`), and handling effects (match `e with`). Abstractions come in two forms: Ocaml abstractions (Λx) and C abstractions (λx). The handler consists of a return case (return `x ⇝ e`), zero or more exception cases (exception `l x ⇝`
Figure 2. Operational semantics of Multicore OCaml effect handlers.
e) with label \( l \), parameter \( x \) and body \( e \), and zero or more effect cases (effect \( l \ x \ k \ \rightarrow \ e \)) with label \( l \), parameter \( x \), continuation \( k \) and body \( e \).

The operational semantics is an extension of the CEK machine semantics [23] for effect handlers, following the abstract machine semantics of Hillerstrom et al. [29]. The key difference from Hillerstrom et al. is that our stacks are composed of alternating sequence of OCaml and C stack segments. The program state is captured as configuration \( C := \{ r, e, o \} \) with the current term \( r \) under evaluation, its environment \( e \) and the current stack \( o \). The term is either an expression \( e \) or a value \( v \). The values are integer constants \( n \), continuations \( k \), closures \( \lambda x . e \), effects being performed (effect \( l \ k \)) and exceptions being raised (exception \( l \)). The environment is a map from variables to values.

The stack \( o \) is either a C stack (\( \gamma \)) or an OCaml stack (\( \omega \)). The C stack \( [\psi, \omega] \) consists of a list of frames \( \psi \), and the OCaml stack \( \omega \) under it. The OCaml stack is either empty \( \\epsilon \) or non-empty \( [k, \gamma] \) with the current continuation \( k \) and the C stack \( \gamma \) under it. Thus, the program stack is an alternating sequence of C and OCaml stacks terminating with an empty OCaml stack \( \bullet \). The frame list \( \psi \) is composed of individual frames \( r \), which is one of an argument frame \( \langle e \rangle \) with the expression \( e \) at the argument position of an application with its environment \( e \), a function frame \( \langle \psi \rangle \gamma \) with the value \( v \) at the function position of an application, and frames for evaluating the arguments of an arithmetic expression.

A continuation \( k \) is either empty \( \\epsilon \) or a non-empty list of fibers. A fiber \( \varphi := \langle \psi, \eta \rangle \) is a list of frames \( \psi \) and a handler closure \( \eta := (h, e) \), which is a pair of handler \( h \) and its environment \( e \). We use the infix operator \( @ \) for appending two lists.

4.2.2 Top-level reductions. The initial configuration for an expression \( e \) is \( \{ r, e, \emptyset \} \) where the environment and the stack are empty. The top-level reductions (Figure 2b) can be performed by either taking a C step \( \rightarrow \) or an OCaml step \( \overline{\overline{\rightarrow}} \).

4.2.3 C reductions. We can take a C step (Figure 2d) by taking an administrative reduction step \( \rightarrow \). The administrative reductions are common to both C and OCaml. The rules VAR, ARITH1, ARITH2, APP1, APP2 and APP3 are standard. ARITH3 performs the arithmetic operation on the integers \( \{n_1 \oplus n_2\} \). RAISE pushes a function frame with exception value to indicate that an exception is being raised. Similarly, PERFORM pushes a function frame with an effect value with an empty continuation \( \{\}, \{\text{return} x \ \rightarrow \ x \}, \emptyset \} \) with no captured frames and an empty handler with an identity return case alone. We shall return to RESUME1 and RESUME2 later.

Continuing with the rest of the C reduction steps, CALLC captures the behaviour of calling a C function. Since the program is currently executing C, we can perform the call on the current stack. In case the abstraction is an OCaml abstraction (CALLBACK), we create an OCaml stack with the C stack as its tail, with the current continuation being empty. This captures the behaviour of calling back into OCaml from C. RetToO returns a value to the enclosing OCaml stack. ExnFwdO forwards a raised exception to the enclosing OCaml stack, unwinding the rest of the frames. This captures the semantics of raising OCaml exceptions from C.

4.2.4 OCaml reductions. In OCaml (Figure 2e), reductions always occur on the top-most fiber in the current stack. AdmInO performs administrative reductions. Callo evaluates an OCaml function on the current stack. ExtCALL captures the behaviour of external calls, which are evaluated on an empty C stack with the current OCaml stack as its tail. RetToC returns a value to the enclosing C stack. In this case, we have exactly one fiber on the stack, and this was created in the rule CALLBACK, whose handler has identity return case alone and the environment is empty. RetFib returns the value from a fiber to the previous one, evaluating the body of the return case.

The rule HANDLE installs a handler by pushing a fiber with no frames and the given handler. The rule ExnHN handles an exception, if the current handler has a matching exception case, unwinding the current fiber. The rule ExnFwdDC forwards the exception to C. Here, there is exactly one fiber on the current stack, and the handler does not have a matching exception case, which we know is the case (see CALLBACK rule). The rule ExnFwdFib forwards the exception to the next fiber if the current handler does not handle it.

The rule EffHn captures the handling of effects when the current handler has a matching effect case. We evaluate the body of the matching case, and bind the continuation parameter \( r \) to the captured continuation \( k' \). Observe that the captured continuation \( k' \) includes the current handler. Intuitively, the handler wraps around captured continuation. This gives Multicore OCaml effect handlers deep handler semantics [29]. EffFwd forwards the effect to the outer fiber, and extends the captured continuation \( k' \) in the process. Recall that we do not capture C frames as part of a continuation. To this end, EffUnHn models unhandled effect. If the effect bubbles up to the top fiber — which we know does not have an effect case (see CALLBACK rule) — we raise Unhandled exception at the point where the corresponding effect was performed. This is achieved by appending the captured continuation to the front of the current continuation.

Observe that continue and discontinue are not part of the expressions. They are encoded as follows: continue \( k \ e = (k \lambda x . \text{Exception} \ l \ x) \) e and discontinue \( k \ l \ e = (k \lambda x . \text{Exception} \ l \ x) \) e. Intuitively, resuming a continuation in both the cases involves evaluating the appropriate abstraction on top of the continuation. We perform the administrative reductions RESUME1 and RESUME2 to evaluate the arguments to continue.
and `discontinue`. The rule `Resume` appends the given continuation to the front of the current continuation, and evaluates the body of the closure.

5 Implementation

We now present the implementation details of effect handlers in Multicore OCaml. We assume an x86_64 architecture for the remainder of this paper. Although Multicore OCaml currently only supports the x86_64 backend on Linux, our design does not preclude other CPUs and OSs.

5.1 Exceptions

The implementation follows the operational semantics, but has a few key representational differences. Unlike the operational semantics, handlers with just exception patterns (exception handlers) are implemented differently than effect handlers. As mentioned in §2.2, exceptions are pervasive in OCaml and are so cheap that they are used for local control flow. Hence, we retain the linked exception handler frame implementation of stock OCaml exception handlers in Multicore OCaml for performance backwards compatibility.

This differs from other research languages with effect handlers [4, 13, 29], which implement exceptions using effects (by ignoring the continuation argument in the handler).

5.2 Heap-allocated fibers

In the operational semantics, the continuations may be resumed more than once. Captured continuations are copies of the original fibers and the continuation copies the fibers and leaves the continuation as it is. In our use cases, continuations will be resumed at-most once, and copying fibers is unnecessary and inefficient. Instead, Multicore OCaml optimises fibers for one-shot continuations. Fibers are allocated on the C heap using `malloc` and are `freed` when the handled computation returns with a value or an exception. Similar to Farvardin et al. [22], we use a stack cache of recently freed stacks in order to speed up allocation.

Figure 3a shows the layout of a fiber in Multicore OCaml. At the bottom of the stack, we have the `handler_info`, which contains the pointer to the parent fiber, and the closures for the value, exception and effect cases. The closures are created by the translation described in §4.1; Multicore OCaml supports exception patterns in addition to effect patterns in the same handler. This is followed by a context block needed for DWARF and GC bookkeeping with callbacks. Then, there is a top-level exception handler frame that forwards exceptions to the parent fiber. When the exceptions are caught by this handler, the control switches to the parent stack, and the exception handler closure `clos_hexn` is invoked. This is followed by the `pc` of the code that returns values to the parent fiber. This stack is laid out such that when the handled computation returns, the control switches to the parent fiber and the value handler `clos_hval` is invoked.

Next, we have the variable-sized area for the OCaml frames. In order to keep fibers small, this area is initially 16 words in length. When the stack pointer `rsp` becomes less than the stack threshold (maintained in the `caml_state` table), the stack is said to have overflowed. On stack overflow, we copy the whole fiber to a new area with double the size. In Multicore OCaml, we introduce stack overflow checks into the function prologue of OCaml functions. These stack overflows are rare and so the overflow checks will be correctly predicted by the CPU branch predictor.

In our evaluation of real world OCaml programs (§6), we observed that most function calls are to leaf functions with small frame sizes. Can we eliminate the stack overflow checks for these functions? To this end, we introduce a small, fixed-sized red zone at the top of the stack. The compiler elides the stack overflow check for leaf functions whose frame size is less than the size of the red zone. The default size of the red zone in Multicore OCaml is 16 words.

Finally, we have the saved exception pointer, which points to the top-most exception frame, and the saved stack pointer, which points to the top of the stack. Switching between fibers only involves saving the exception and the stack pointer of the current stack and loading the same on the target stack. Since OCaml does not generate pointers into the stack, the two `fiber_info` fields are the only ones that need to be updated when fibers are moved.

5.3 External calls and callbacks

Since C functions do not have stack overflow checks, we have to execute the external calls in the system stack. Calling a C function from OCaml involves saving the stack pointer in the current fiber, saving the allocation pointer value in `r15` in the `caml_state`, updating `rsp` to the top of system stack (maintained in `caml_state`), and calling the C function. The actions are reversed when returning from the external call. For C functions that take arguments on the stack, the arguments must be copied to the C stack from the OCaml stack.

When we first enter OCaml from C, a new fiber is allocated for the main OCaml stack. Since callbacks may be frequent in OCaml programs that use finalisers, we run the callbacks on the same fiber as the current one. For example, the layout of the Multicore OCaml stack at `caml_raise_exn` in the `meander` example from §2 is shown in Figure 3b. The functions `caml_call_c` and `caml_call_ocaml` switch the stacks, and hence are shown in both the system stack and the fiber. Since we are reusing the fiber for the callback, care must be taken to save and restore the `handler_info` before calling and after returning from `c_to_ocaml` function, respectively. Thanks to the fiber representation, external calls and callbacks remain competitive with stock OCaml.

5.4 Effect handlers

In order to illustrate the implementation of handling effects, consider the example presented in Figure 3c. The layout of the program state as the program executes is captured in
Figure 3d. The code performs effect \( E \) which is handled in the outer-most handler, and is immediately resumed. The arrows between the fibers are parent pointers. At position \( p_3 \), \( \text{rsp} \) is at the top of the fiber \( f \).

When the effect \( E \) is performed, we allocate a continuation object \( \text{ke} \) in the OCaml heap that points to the current fiber \( f \), set fiber \( f \)'s parent pointer to \( \text{NULL} \), and evaluate the continuation closure \( \text{clos_heffect} \) on the parent fiber \( e \) with the effect \( E \) and the continuation \( \text{ke} \) as arguments. Since the first handler does not handle effect \( E \), the effect is reperformed (§4.1) by appending the fiber \( e \) to the tail of continuation \( \text{ke} \), set fiber \( e \)'s parent pointer to \( \text{NULL} \), and evaluate the current continuation closure on the parent fiber \( \text{main} \) with \( E \) and \( \text{ke} \) as arguments, which handles \( E \) (position \( p_2 \)). Thus, continuations are captured without copying frames. Since every handler closure is evaluated until a matching one is found, the time taken to handle an effect is linear in the number of handlers. We observed that the handler stack is shallow in real programs.

When the continuation is resumed, we overwrite the value of \( \text{ke} \) to \( \text{NULL} \) to enforce at-most once semantics. Resuming a continuation involves traversing the linked-list of fibers and making the last fiber point to the current fiber. Just as in the operational semantics, the implementation invokes the appropriate closure to either continue or discontinue the continuation (position \( p_3 \)). We perform tail-call optimisation so that resumptions at tail positions do not build up stack.

### 5.5 Stack unwinding

The challenge with DWARF stack unwinding is to make it aware of the non-contiguous stacks. While the complete details of DWARF stack unwinding is beyond the scope of the paper, it is beneficial to know how DWARF unwind tables are constructed in order to appreciate our solution. We refer the interested reader to Bastian et al. [3] for a good overview of DWARF stack unwinding.

Logically, DWARF call-frame information maintains a large table which records for every machine instruction where the return address and callee-saved registers are stored. To avoid reifying this large table, DWARF directives represent the table using a compact bytecode representation. This representation is interpreted on demand for building the stack unwind table. Within each function, DWARF maintains a canonical frame address (CFA) and is traditionally the stack pointer before entering this function. Hence, on x86-64, where the return address is pushed on the stack on call, the return address is at CFA - 8.

The goal is to encode the CFA using DWARF directives when stacks are switched. Recall that stack switching occurs in effect handlers, external calls and callbacks. When starting to run on a new fiber, we insert DWARF bytecode to follow the parent_fiber pointer and dereference the saved_sp to get the CFA (saved_sp + 8). During callbacks into OCaml, we save the current system stack pointer in the context block in Figure 3a to identify the CFA in the C stack. DWARF unwinding for external calls is implemented by following a link to the current OCaml stack pointer. With these changes, we get the same backtrace for the \( \text{meander} \) program from §2.3, modulo runtime system functions due to effect handlers.

We have utilized the DWARF verification tool from [3] to flag possible errors in the hand written DWARF directives where we have altered the OCaml assembler to implement...
effect handlers. In three places additional manual verification was required; two were where the stack is intentionally switched between fibers and one was where the control returns to the parent fiber which happens to be resused due to effect handler invocation, which the tool could not track.

Despite the correct DWARF unwind information, using DWARF to record call stack information in perf only captures the call stack of the current fiber in Multicore OCaml. Since stack unwinding using DWARF is slow due to bytecode interpretation overhead, perf dumps the (user) call stack when sampled [3]. This only includes the frames from the current fiber. This is a limitation of perf and not of our stack layout.

5.6 Garbage collection

Recall that OCaml programs are written with the expectation that function calls return exactly once (§3). Consider the scenario when a continuation is neither continued nor discontinued. Since fibers are malloced, and are freed when the computation being handled returns a value or raises an exception, not resuming continuations leaks memory. In addition, unresolved continuations may also leak other resource such as open file descriptors.

We consider well-behaved Multicore OCaml programs to resume their continuations exactly once. This can be enforced by installing a finaliser on every captured continuation that calls discontinue on the continuation during finalisation and ignores the result:

```
gc.finalise (fun k ->
  try ignore (discontinue k Unwind) with _ -> () ) k
```

However, Multicore OCaml does not enforce this behaviour in order to avoid the costs associated with installing a finaliser every time a continuation is captured.

For stack walking during GC, the only change necessary was to make the runtime aware of fiber layout and how to follow the parent pointers. The stack map generation and scanning each frame for roots remains the same. The challenges and solutions for integrating fibers with the concurrent mark-and-sweep GC of Multicore OCaml have been discussed previously [54].

6 Evaluation

In this section, we evaluate the performance of Multicore OCaml effect handlers against the performance requirements set in §1.4. Multicore OCaml is an extension of the 4.10.0 OCaml compiler with support for shared memory parallelism and effect handlers. These results were obtained on a 2-socket Intel®Xeon®Gold 5120 x86-64 [31] server running Ubuntu 18.04 with 64GB of main memory.

6.1 No effects benchmarks

In this section, we measure the impact of the addition of effect handlers on code that does not use effect handlers. Our macro benchmark suite consists of 54 real OCaml workloads including verification tools (Coq, Cubicle, AltErgo), parsers (menhir,
For callbacks, since we reuse the current fiber stack, we need to ensure it has enough room for inserting additional frames, while stock does not need to do this. Callback performance is less important than external calls, which are far more numerous. As seen previously, these results however do not affect macro benchmark performance.

### 6.2 No perform benchmarks

Next, we aim to quantify the overhead of setting up and tearing down effect handlers compared to a non-tail function call. To this end, we surround the non-tail calls in the recursive micro benchmarks with an effect handler. These programs do not perform effects. We also implemented the same benchmarks using a concurrency monad [12] (monad) as a proxy for CPS versions. Recall that the OCaml compiler does not use CPS in its IR. In the monad version, we use a fork to invoke the non-tail call and use an MVar to collect its result.

The results are presented in Table 2. They show that using effect handlers (concurrency monad) is $10.02 \times (67.09 \times)$ more expensive than the idiomatic implementation using non-tail calls. The concurrency monad suffers due to the heap allocation of continuation frames (which need to be garbage collected), whereas effect handlers benefit from stack allocation of the frames. For example, the number of major collections for the ack benchmark is 0 for stock OCaml, 1 for MC and 112 for monad. Our concurrency monad (and other monadic concurrency libraries such as Lwt [43] and Async [1]) also have other downsides – exceptions, backtraces, and DWARF unwinding are no longer useful due to the lack of a stack.

We note that a compiler that uses CPS IR will be faster than the concurrency monad implementation due to optimisations to reduce the heap allocation of continuation frames. But Farvardin et al. [22] show that CPS with optimisations is still slower than using the call stack.

### 6.3 Concurrent benchmarks

Next we look at benchmarks that utilize non-local control flow using effect handlers. First, we quantify the overhead of individual operations in effect handling. Consider the following annotated code:

```ocaml
effect E : unit
effects
| e -> E e
| E k -> E k

The sequence a-b involves allocating a new fiber and switching to it. b-c is performing the effect and handling it. c-d is resuming the continuation. d-e is returning from the fiber with a value and freeing the fiber. We measured the time taken to execute these sequences using `perf` support for cycle-accurate tracing on modern Intel processors. We executed 10 iterations of the code, with 3 warm-up runs. We observed that the sequences a-b, b-c, c-d and d-e took 23 ns, 5 ns, 11 ns and 7 ns, respectively. For reference, the idle memory load latency for the local NUMA node is 93.2 ns as measured using the Intel MLC tool [46].

Generators. Generators allow data structures to be traversed on demand. Many languages including JavaScript and Python provide generators as a built-in primitive. Using effect handlers (MC), given any data structure (`a t`) and its iterator (val iter: `a t -> ('a -> unit) -> unit`), we can derive its generator function (val next: unit -> 'a option). The code is available in the supplementary material. We evaluate the performance of traversing a complete binary tree of depth 25 using this generator. This involves 2^26 stack switches in total. For comparison, we implemented a handwritten, selective CPSed [48], defunctionalised [17] version (cps) and a concurrency monad (monad) version of the generator for the tree. Both cps and monad versions are specialised to the binary tree with the usual caveats of not using the stack for function calls. We observed that the cps version was the fastest, thanks to specialisation and hand optimisation. MC version was only 2.76× slower than cps while being a generic solution, and the monad version was 8.69× slower than cps.

Chameneos. Chameneos [34] is a concurrency game aimed at measuring context switching and synchronization overheads. Our implementation uses MVars for synchronization. We compare effect handler (MC), concurrency monad (monad) and Lwt, a widely used concurrency programming library for OCaml (lwt) versions. We observed that MC was the fastest, and monad (lwt) was 1.67× (4.29×) slower than MC.

Web server. As one of our primary goals of adding effect handlers to OCaml is to support direct-style concurrency, we implement a full-fledged HTTP/1.1 web server using effect handlers (MC). We use effect handlers to implement a user-level threading library that spawns a thread per request. We use `httpaf` [30] for HTTP handling, and `libev` [41] for the eventloop. We compare our implementation against an Lwt version (lwt) which also uses `httpaf` and `libev`. Unlike using effect handlers, the Lwt version is written in monadic style and does not have the notion of a thread per request. For comparison, we include a Go 1.13 version (go) that uses the net/http [47] package. As both the OCaml versions are single threaded, the Go benchmark is run with GOMAXPROCS=1.

The client workload was generated with wrk2 [60]. We maintain 1k open connections and perform requests for a static web page at different rates, and record the service rate and latency. The throughput and tail latency graphs are given in Figures 4a and 4b. In all the versions, the throughput plateaus at around 30k requests per second. We measure the tail latencies at 2/3rd of this rate (20k requests per second) to simulate optimal load. We observe that both of the OCaml

### Table 2. Micro benchmarks with handlers but no perform.

<table>
<thead>
<tr>
<th></th>
<th>MC</th>
<th>fib</th>
<th>motzkin</th>
<th>sudan</th>
<th>tak</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC</td>
<td>12.25</td>
<td>12.05</td>
<td>11.44</td>
<td>6.74</td>
<td>8.9</td>
</tr>
<tr>
<td>monad</td>
<td>348.69</td>
<td>69.77</td>
<td>39.24</td>
<td>33.29</td>
<td>42.79</td>
</tr>
</tbody>
</table>

Each entry is the slowdown factor (× times) over its idiomatic implementation in stock OCaml.
versions remain competitive with go, and MC performs best in terms of tail latency.

Multicore OCaml supports backtraces for continuations in addition to backtraces of the current stack as in stock OCaml. Using effect handlers in a system such as a web server aides debugging and profiling because it is possible to get a backtrace snapshot of all current requests. Such a feature is available in Go [27], but not in OCaml concurrency libraries such as Async and Lwt which lack the notion of a thread.

7 Related Work

There are several strategies for implementing effect handlers. Eff [4], Helium [7], Frank [13] and the Links server backend [29] use an interpreter similar to our operational semantics to implement effect handlers. Effekt [52], Links JavaScript backend [29] and Koka [38] use type-directed selective CPS translation. These language are equipped with an effect system, which allows compiling pure code in direct-style and effectful code in CPS. Leijen [39] implements effect handlers as a library in C using stack copying. C allows pointers into the stack, so care is taken to ensure that when continuations are resumed, the constituent frames are restored to the same memory addresses as at the time of capture. Kiselyov et al. [37] use an implementation of multi-prompt delimited continuations as an OCaml library [36] to embed the Eff language in OCaml. Indeed, Forster et al. [26] showed that in an untyped setting, effect handlers, monadic reflection and delimited control can macro-express each other.

Multicore OCaml uses the call stack for implementing continuations (as do [36, 39]), but with one-shot continuations. Bruggeman et al. [10] show how to implement one-shot continuations efficiently using segmented stacks in Scheme. Farvardin et al. [22] perform a comprehensive evaluation of various implementation strategies for continuations on modern hardware. Multicore OCaml stacks do not neatly fit the description of one of these implementation strategies – they are best described as using the reserve strategy from Farvardin et al. for each of the fibers, which are linked to represent the current stack and the captured continuations.

Kawahara et al. [35] implement one-shot effect handlers using coroutines as a macro-expressive translation, and present an embedding in Lua and Ruby. Lua provides asymmetric coroutines [18] where each coroutine uses its own stack similar to how each handled computation runs in its own fiber in Multicore OCaml.

Multicore OCaml is not the first language to support stack inspection in the presence of non-local control operators. Chez Scheme supports continuation marks [24] which permit stack inspection as a language feature. This enables implementation of dynamic binding, exceptions, profilers, debuggers, etc, in the presence of first-class continuations. As the authors note, continuation marks can be implemented using effect handlers, but direct support for continuation marks leads to better performance. In this work, we focus on retaining the support for stack inspection through DWARF unwind tables in the presence of effect handlers.

The interaction of non-local control flow and resources has been studied extensively. Scheme uses dynamic-wind [51], which is a generalisation of Common Lisp unwind-protect [56], which ensures de-allocation (and re-allocation) of resources every time the non-local control leaves and enters back into a context. dynamic-wind is not quite sufficient for resource cleanup up [21, 53] as returning exits must be distinguished from non-returning ones, and resource cleanup is only necessary during non-returning exits.

Multicore OCaml builds on the existing defensive coding practices against exceptions to clean up resources on non-returning exits. Our continuations are assumed to be resumed exactly once using continue or discontinue. As a result, when a computation performs an effect, we are guaranteed that the control will return. For the non-returning cases (value and exceptional return), the code already handles resource cleanup. OCaml does not have a try/finally construct commonly used for resource cleanup in many programming languages. Alternative standard libraries such as Base [2] and Core [15] provide mechanisms analogous to unwind-protect, which are implemented using exceptions. Thus, the linear use of continuations enabled by the discontinue primitive ensures backwards compatibility of legacy OCaml systems code under non-local control flow introduced by effect handlers. Leijen [40] explicitly extends effect handlers with initially and finally clauses in Koka for resource safety. Dolan et al. [19] describe the interaction of effect handlers and asynchronous exceptions. This is orthogonal to the contributions of this paper. Our focus is the compiler and runtime system support for implementing effect handlers.

8 Conclusions

Our design for effect handlers in OCaml walks the tightrope of maintaining compatibility (for profiling, debuggers and negligible overheads for existing programs), while unlocking the full power of non-local control flow constructs. Our evaluation shows that we have achieved our goal: we retain compatibility with a surprisingly low performance overhead for sequential code that preserves the spirit of “fast exceptions” that has always characterised OCaml programming. We believe that the introduction of effect handlers into OCaml
implemented using lightweight fibers, along with a parallel runtime [54], will open OCaml to highly scalable concurrent and task-parallel applications with minimal hit to sequential performance.
References

janestreet.com/async/


1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650

28th International Conference on Parallel Architectures and Compilation Techniques (PACT). 29–42. https://doi.org/10.1109/PACT.2019.00011


[64] HTTP client and server implementations in Go. https://golang.org/pkg/net/http/

Supplementary Material

No effect macro benchmarks

Figures 1 and 2 present the impact on macro benchmarks that do not use effects using the Multicore OCaml compiler which supports effect handlers. The aim is to quantify the impact of the addition of effect handlers to the language on code that does not use effect handlers. The \( \text{MC} \) is Multicore OCaml with the default red zone size (16 words). \( \text{MC+RedZone0} \) uses red zone size of 0; every OCaml function has a stack overflow check. \( \text{MC+RedZone32} \) uses a red zone size of 32 words.

Experimental support for multi-shot continuations

Languages with effect handlers typically allow continuations to be resumed more than once to enable use cases such as backtracking search and probabilistic programming (for example, see Uber’s Pyro language [8]). Multicore OCaml has an experimental feature for explicitly cloning continuations, which makes a copy of each of its constituent fibers.

Multi-shot continuations are difficult to reason about in the presence of linear resources such as file descriptors. Recall that OCaml programs are written with the expectation that function calls return exactly-once either normally or exceptionally. With multi-shot continuations, they may return more than once, leading to errors such as double-freeing malloced memory. Since our goal is to retrofit effect handlers onto OCaml, we relegate continuation cloning to the OCaml \( \text{Obj} \) module, which contains unsafe features.

In addition, multi-shot continuations currently are not safe under compiler optimisations. For example consider the following program:

```ocaml
effect E : unit
let foo () = perform E
let bar f =
  let r = ref 0 in
  f ();
  r := !r + 1;
  !r
;;
match bar foo with
| v -> v
| effect E k ->
  let k' = Obj.clone_continuation k in
  let r1 = continue k () in
  let r2 = continue k' () in
  Printf.printf "%d %d\n" r1 r2;
  0
```

This program is expected to print 1 2. However, OCaml optimizes \( \text{bar} \) by converting the heap allocation of \( r \) to a stack allocation. As a result, each continuation ends up with a copy of \( r \), and the program prints 1 1. Thus, in the presence of multi-shot continuations, compiler optimisations need to be made aware of the effects of a function. We leave proper support for multi-shot continuations to future work.
Figure 1. Normalized time of macro benchmarks. Baseline is Stock OCaml, whose running time in seconds in given in parenthesis.
Figure 2. Normalized OCaml text section size (OTSS) of macro benchmarks. Baseline is Stock OCaml, whose size in bytes given in parenthesis.