

# Effective Concurrency through Algebraic Effects

Stephen Dolan<sup>1</sup>, Leo White<sup>2</sup>, KC Sivaramakrishnan<sup>1</sup>, Jeremy Yallop<sup>1</sup>, and Anil Madhavapeddy<sup>1</sup>

<sup>1</sup>University of Cambridge

<sup>2</sup>Jane Street Capital

*Algebraic effects and handlers provide a modular abstraction for expressing effectful computation, allowing the programmer to separate the expression of an effectful computation from its implementation. We present an extension to OCaml for programming with linear algebraic effects, and demonstrate its use in expressing concurrency primitives for multicore OCaml.*

## 1 Motivation

Multicore-capable functional programming language implementations such as Glasgow Haskell Compiler, F#, Manticore and MultiMLton expose one or more libraries for expressing concurrent programs. The concurrent threads of execution instantiated through the library are in turn multiplexed over the available cores for speed up. A common theme among such runtimes is that the primitives for concurrency along with the concurrent thread scheduler is baked into the runtime system. Over time, the runtime system itself tends to become a complex, monolithic piece of software, with extensive use of locks, condition variables, timers, thread pools, and other arcana. As a result, it becomes difficult to maintain existing concurrency libraries, let alone add new ones. Such lack of malleability is particularly unfortunate as it prevents developers from experimenting with custom concurrency libraries and scheduling strategies, preventing innovation in the ecosystem. Our goal with this work is to provide a minimal set of tools with which programmers can implement new concurrency primitives and schedulers as OCaml libraries.

## 2 A Taste of Effects

Let us illustrate the algebraic effect extension in multicore OCaml by constructing a concurrent round-robin scheduler with the following interface:

```
(* Control operations on threads *)
val fork : (unit -> unit) -> unit
val yield : unit -> unit
(* Runs the scheduler. *)
val run : (unit -> unit) -> unit
```

The basic tenet of programming with algebraic effects is that performing an effectful computation is separate from its interpretation [1, 5]. In particular, the interpretation is dynamically chosen based on the context in which an effect is performed. In our example, spawning a new thread and yielding control to another are effectful actions, for which we declare the following effects:

```
type _ eff +=
| Fork : (unit -> unit) -> unit eff
| Yield : unit eff
```

The type `'a eff` is the predefined extensible variant type for effects, where `'a` represents the return type of performing the effect. For convenience, we introduce new syntax using which the same declarations are expressed as follows:

```
effect Fork : (unit -> unit) -> unit
effect Yield : unit
```

Effects are performed using the primitive `perform` of type `'a eff -> 'a`. We define the functions `fork` and `yield` as follows:

```
let fork f = perform (Fork f)
let yield () = perform Yield
```

What is left is to provide an interpretation of what it means to perform `fork` and `yield`. This interpretation is provided with the help of *handlers*.

```
1 let run main =
2   let run_q = Queue.create () in
3   let enqueue k = Queue.push k run_q in
4   let rec dequeue () =
5     if Queue.is_empty run_q then ()
6     else continue (Queue.pop run_q) ()
7   in
8   let rec spawn f =
9     match f () with
10    | () -> dequeue ()
11    | exception e ->
12      print_string (to_string e);
13      dequeue ()
14    | effect Yield k ->
15      enqueue k; dequeue ()
16    | effect (Fork f) k ->
17      enqueue k; spawn f
18  in
19  spawn main
```

The function `spawn f` (line 8) evaluates `f` in a new thread of control. `f` may return normally with value `()` or exceptionally with an exception `e` or effectfully with the effect performed along with the delimited continuation `[4] k`. In the pattern `effect e k`, if the effect `e` has type `'a eff`, then the delimited continuation `k` has type `('a, 'b) continuation`, i.e., the return type of the effect `'a` matches the argument type of the continuation, and the return type of the delimited continuation is `'b`.

Observe that we represent the scheduler queue with a queue of delimited continuations, with functions to manipulate the queue (lines 2–6). In the case of normal or exceptional return, we pop the scheduler queue and resume the resultant continuation using the `continue` primitive (line 6). `continue k v` resumes the continuation `k` : `('a, 'b) continuation` with value `v` : `'a` and returns a value of type `'b`. In the case of effectful return with `Fork f effect` (lines 16–17), we enqueue the current continuation to the scheduler queue and spawn a new thread of control for evaluating `f`. In the case of `Yield effect` (lines 14–15), we enqueue the current continuation, and resume some other saved continuation from the scheduler queue.

### 3 Implementing Algebraic Effects

The main challenge in the implementation of algebraic effects is the efficient management of delimited continuations. In multicore OCaml [3], the delimited continuations are implemented using *fibers*, which are small heap-allocated, dynamically resized stacks. Fibers represent the unit of concurrency in the runtime system.

Our continuations are linear (one-shot) [2], in that once captured, they may be resumed at most once. Capturing a one-shot continuation is fast, since it involves only obtaining a pointer to the underlying fiber, and requires no allocation. OCaml uses a calling convention without callee-save registers, so capturing a one-shot continuation requires saving no more context than that necessary for a normal function call.

Since OCaml does not have linear types, we enforce the one-shot property at runtime by raising an exception the second time a continuation is invoked. For applications requiring true multi-shot continuations (such as probabilistic programming [6]), we envision providing an explicit operation to copy a continuation.

While continuation based concurrent functional programming runtimes such as Manticore and MultiMLton use undelimited continuations, our continuations are delimited. We believe delimited continuations enable complex nested and hierarchical schedulers to be expressed more naturally due to the fact that they introduce parent-

child relationship between fibers similar to a function invocation.

## 4 Running on Multiple Cores

Multicore OCaml provides support for shared-memory parallel execution. The unit of parallelism is a *domain*, each running a separate system thread, with a relatively small local heap and a single shared heap shared among all of the domains. In order to distributed the fibers amongst the available domains, work sharing/stealing schedulers are initiated on each of the domains. The multicore runtime exposes to the programmer a small set of locking and signalling primitives for achieving mutual exclusion and inter-domain communication.

The multicore runtime has the invariant that there are no pointers between the domain local heaps. However, the programmer utilising the effect library to write schedulers need not be aware of this restriction as fibers are transparently promoted from local to shared heap on demand.

## References

- [1] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *The Journal of Logic and Algebraic Programming*, 84(1):108–123, 2015.
- [2] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing Control in the Presence of One-shot Continuations. In *PLDI*, 1996.
- [3] S. Dolan, L. White, and A. Madhavapeddy. Multicore OCaml. In *OCaml Users and Developers Workshop*, 2014.
- [4] A. Filinski. Representing Monads. In *POPL*, 1994.
- [5] O. Kammar, S. Lindley, and N. Oury. Handlers in Action. In *ICFP*, 2013.
- [6] O. Kiselyov and C.-C. Shan. Embedded probabilistic programming. In *Domain-Specific Languages*, 2009.