Effectively Tackling the Awkward Squad

Stephen Dolan Spiros Eliopoulos Daniel Hillerström Anil Madhavapeddy
KC Sivaramakrishnan Leo White

Useful real-world functional programs must often confront the *Awkward Squad* [7], a range of *un-beautiful* issues concerning the interplay between concurrency, input/output, exceptions, resources, etc. We show that algebraic effects and their handlers can elegantly express such programs without compromising performance. In particular, we introduce *asynchronous effects* and their handlers, and show how they elegantly solve the interaction between user-level threads and operating system services.

1 Introduction

Algebraic effects and their handlers have been steadily gaining attention as a programming language feature for composably expressing user-defined computational effects. While several prototype implementations of languages incorporating algebraic effects exist, Multicore OCaml [2] incorporates effect handlers as the primary means of expressing concurrency in the language [3][1]. The modular nature of effect handlers allows the concurrent program to abstract over different scheduling strategies. Moreover, effect handlers allow concurrent programs to be written in *direct-style* retaining the simplicity of sequential code as opposed to callback-oriented style with either monadic concurrency libraries such as Lwt [8] and Async [5] for OCaml or explicit callbacks.

2 Event-based I/O in direct-style

In a language with lightweight threads (*fibers* as we call them in Multicore OCaml), invoking blocking function calls such as `Unix.accept` would block the entire scheduler, preventing other threads to run. The standard solution to this problem is to use an *event loop*, suspending each task performing a blocking I/O operation, and then multiplexing the outstanding I/O operations through an OS-provided blocking mechanism such as `select`, `epoll`, `kqueue`, `IOCP` (IO Completion Port), etc. Such asynchronous, non-blocking code typically warrants callback-oriented programming, making the continuations of I/O operations explicit through explicit callbacks (à la JavaScript) or concurrency monad (Lwt and Async libraries for OCaml). This warrants a wholesale departure from synchronous direct-style code. The resultant code is arguably messier, though monadic concurrency libraries do have the benefit of automatic mutual exclusion: context switches only occur at bind points.

Effect handlers lets us retain the direct-style while still allowing the use of event loops. We would declare an effect for an `accept` function `effect Accept : file_descr → (file_descr * sockaddr)` with the handler:

```
| effect (Accept fd) k →
    if poll_rd fd then
      try continue k (Unix.accept fd)
      with e → discontinue k e
    else (block_accept fd k; run_next ())
```

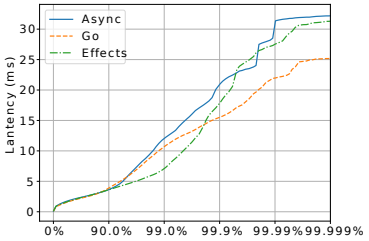

Figure 1: Latency profile of client requests. 1k connections, 10k requests/sec.

We first poll the file descriptor `fd` to see whether it is available to read. If so, we immediately perform the blocking call (which is expected to succeed[2]), and resume the fiber with the result. However, if the call would block, then we record that the fiber is waiting to accept on `fd` and switch to the next fiber from the scheduler queue as follows:

```
let run_next () =
  if Queue.is_empty run_q then
    if io_is_pending () then begin
      wait_until_io_ready ();
      do_io ();
      run_next ()
    end else () (* done *)
  else Queue.pop run_q ()
```

`run_next` function first runs all the available threads, and then if any I/O is pending it waits until at least one of the I/O operations is ready. Then, it tries to perform the I/O and continue. If the scheduler queue is empty, and there are no pending I/O, then the scheduler returns. In a similar fashion we can implement asynchronous variants of `Unix.send` and `Unix.recv`. Using this API, we can write a simple server that echoes client messages until client goes away as follows:

```
let rec echo_server sock =
  let sent = ref 0 in
  let msg_len = (* receive message *)
    try recv sock buffer 0 buf_size [] with
    | _ → 0 (* Treat exceptions as 0 length msg *)
  in
  if msg_len > 0 then begin
    (* echo message *)
    (try while !sent < msg_len do
      let write_count =
        send sock buffer !sent (msg_len - !sent) [] in
      sent := write_count + !sent
    done with _ → ()); (* ignore send failures *)
    echo_server sock
  end else close sock (* client left, close conn *)
```

The details of the code are not important, but observe that the code is in direct-style and moreover it is the *same* code

[1] We refer the interested readers to the full version of the paper for a primer on effect handlers in Multicore OCaml: `http://kcsrk.info/papers/system_effects_may_17.pdf`

[2] In reality, the call might not succeed due to a variety of exceptional cases that must be handled for correctness [1]. But importantly, the client-facing API remains in a direct-style.

for the synchronous, blocking echo server. In similar vein, we have implemented an effect-based asynchronous I/O library, `aeio` [1], that exposes a direct-style API to the clients. Experimental results (Fig. 1) of using the library for as a backend for `httpaf`, a full-featured OCaml web server demonstrates that effect handlers perform on par with highly optimised monadic concurrency libraries, while retaining the simplicity of direct-style code.

3 Asynchronous effects

Effect handlers thus provide a nice abstraction for expressing user-level cooperative fiber schedulers. However, if a fiber runs a long running pure computation, it leads to an undesirable situation where the fiber hogs the core and does not let other fibers run until the pure computation is complete. The standard solution is to preempt the fiber periodically by installing an interval timer. However, with a user-level scheduler, it is unclear how the signal handler for the timer interrupt (which itself is an asynchronous computation) could get hold of the continuation of the main computation. One could envision supplying the signal handler function with the current continuation, but this makes the API more awkward [6].

We make the observation that various asynchronous *upcalls* from the OS such as timer expiry, interrupts, signals, I/O completion notifications can be treated as asynchronous effects. Rather than handling the timer interrupt in a signal handler, the runtime raises an effect `TimerTick` in the current fiber, which can be handled like a synchronous `Yield` effect to implement preemptive scheduling:

```
| effect TimerTick k →
  (* add current fiber to scheduler *)
  enqueue (fun () → continue k ());
  run_next()
```

If the `TimerTick` is unhandled, it is a noop.

3.1 Signal handling

Signals are also modeled as asynchronous effects. On Unix-like systems, when the user of a command-line program presses `Ctrl-z`, the `SIGTSTP` signal is sent to the running program. By default, this suspends the program. However, we might want a different behaviour for `Ctrl-z` where we preempt the running fiber if we are in the scheduler, and suspend the program otherwise. We can achieve this by handling the asynchronous `Suspend` event in the scheduler to handle the `SIGTSTP` signal:

```
match f () with
| v → ...
| ....
| effect Suspend k → (* handle SIGTSTP *)
    enqueue (fun () → continue k ());
    run_next()
```

This implementation, however, has a subtle bug; if `Ctrl-z` was received while the control is in the body of clause for the `Suspend` effect, then the program is suspended. This is not the intended behaviour as we are in the middle of preempting a thread, and we do not expect to be suspended. In order to eliminate this possibility we require a means for temporarily disabling asynchronous exceptions. We follow the good advice of Marlow et al. in the design of GHC Haskell's asynchronous exceptions [4], and prefer *scoped masking combinators*:

```
mask (fun () →
  match unmask f with
  | v → ...
  | ...
  | effect Suspend k →
      enqueue (fun () → continue k ());
      run_next())
```

The changes to the masking state made by `mask` and `unmask` apply only to one scope, and are automatically updated when entering and leaving the scope. Scoped combinators are also exception safe: if `run_next` raises an exception which escapes the handler scope, then the masking state is automatically reset to that of the parent scope.

3.2 Asynchronous I/O notifications

Operating systems provide a number of different interfaces with which to perform I/O. The simplest is the direct-style *blocking I/O*, in which the program calls I/O functions provided by the operating system, which do not return until the operation completes. This allows a straightforward style of programming in which the sequence of I/O operations matches the flow of the code. We aim to preserve this style of programming, but implement it using alternative operating system interfaces that allow multiple I/O operations to be overlapped.

Earlier we saw one way of accomplishing this with effects, by using operating system multiplexing mechanisms like `select`, `poll`, etc., which block until one of several file descriptors is ready. An alternative interface is *asynchronous I/O*, in which multiple operations are submitted to the operating system, which overlaps their execution. However, applications written using asynchronous I/O tend to have complex control flow which does not clearly explain the logic being implemented, due to the complexity of handling the operating system's asynchronous notifications of I/O completion.

We propose effects and handlers as a means of writing direct-style I/O code, but using the asynchronous operating system interfaces. We introduce two new effect operations: `Delayed`, which describes an operation that has begun and will complete later, and `Completed`, which describes its eventual completion. Both of these take an integer parameter, which is an ID number identifying the particular operation.

Potentially long-running operations like `read` perform the `Delayed` effect, indicating that the operation has been submitted to the operating system but has not yet completed. Later, upon receipt of an operating-system completion notification, the asynchronous effect `Completed` is performed.

Using this mechanism, support for asynchronous completions can be added to the scheduler by adding clauses for the `Delayed` and `Completed` effects, where `ongoing_io` is an initially empty hash table:

```
| effect (Delayed id) k →
  Hashtbl.add ongoing_io id k
| effect (Completed id) k →
  let k' = Hashtbl.find ongoing_io id in
  Hashtbl.remove ongoing_io id;
  enqueue (fun () → continue k ());
  continue k' ()
```

In this sample, the continuation `k` of the `Delayed` effect is the continuation of the code performing the I/O operation, which instead of being immediately invoked is stored in a hash table until it can be invoked without blocking.

The continuation `k` of the `Completed` effect is the continuation of whichever fiber was running when the I/O completed. This scheduler chooses to preempt that fiber in favour of the fiber that performed the I/O, by retrieving the continuation `k'` from the hashtable and continuing it. Equally, the scheduler's policy could be to give priority to the already running fiber, by swapping `k` and `k'` in the last two lines.

References

[1] Aeio: An asynchronous, effect-based I/O library, 2017. Accessed: 2017-05-05 09:21:00.

[2] S. Dolan, L. White, and A. Madhavapeddy. Multicore OCaml. OCaml Workshop, 2014.

[3] S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency through algebraic effects. OCaml Workshop, 2015.

[4] S. Marlow, S. P. Jones, A. Moran, and J. Reppy. Asynchronous exceptions in Haskell. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 274–285, New York, NY, USA, 2001. ACM.

[5] Y. Minsky, A. Madhavapeddy, and J. Hickey. *Real World OCaml - Functional Programming for the Masses*. O'Reilly, 2013.

[6] Signal Handling in MLton Threads. `http://www.mlton.org/MLtonSignal`, 2016. accessed: 1-Jun-2017.

[7] S. Peyton Jones. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, pages 47–96. IOS Press, January 2001.

[8] J. Vouillon. Lwt: A cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 3–12, New York, NY, USA, 2008. ACM.