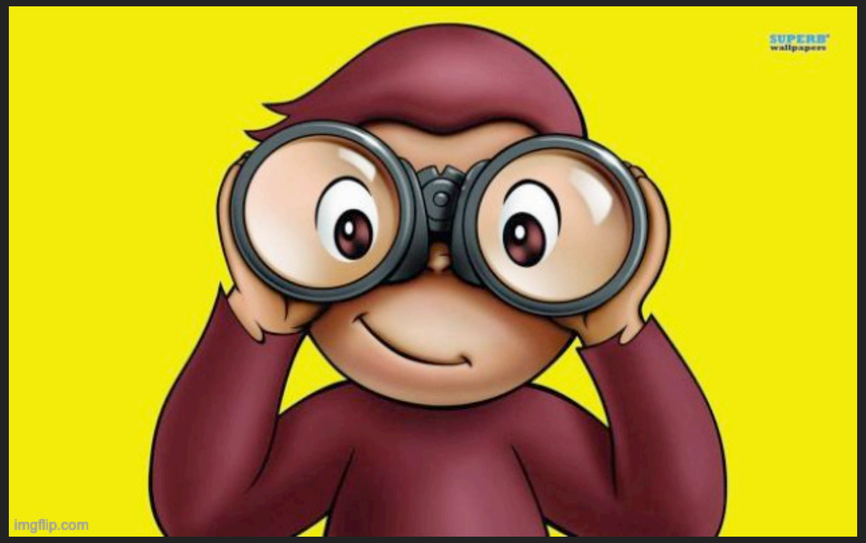
A SURVEY

ALGEBRAIC EFFECTS


OCaml


Tarides

Algebraic effects and their handlers have been steadily gaining attention as a programming language feature for composably expressing user-defined computational effects


SUPERB
wallpapers
imgflip.com

KOKA

EFF

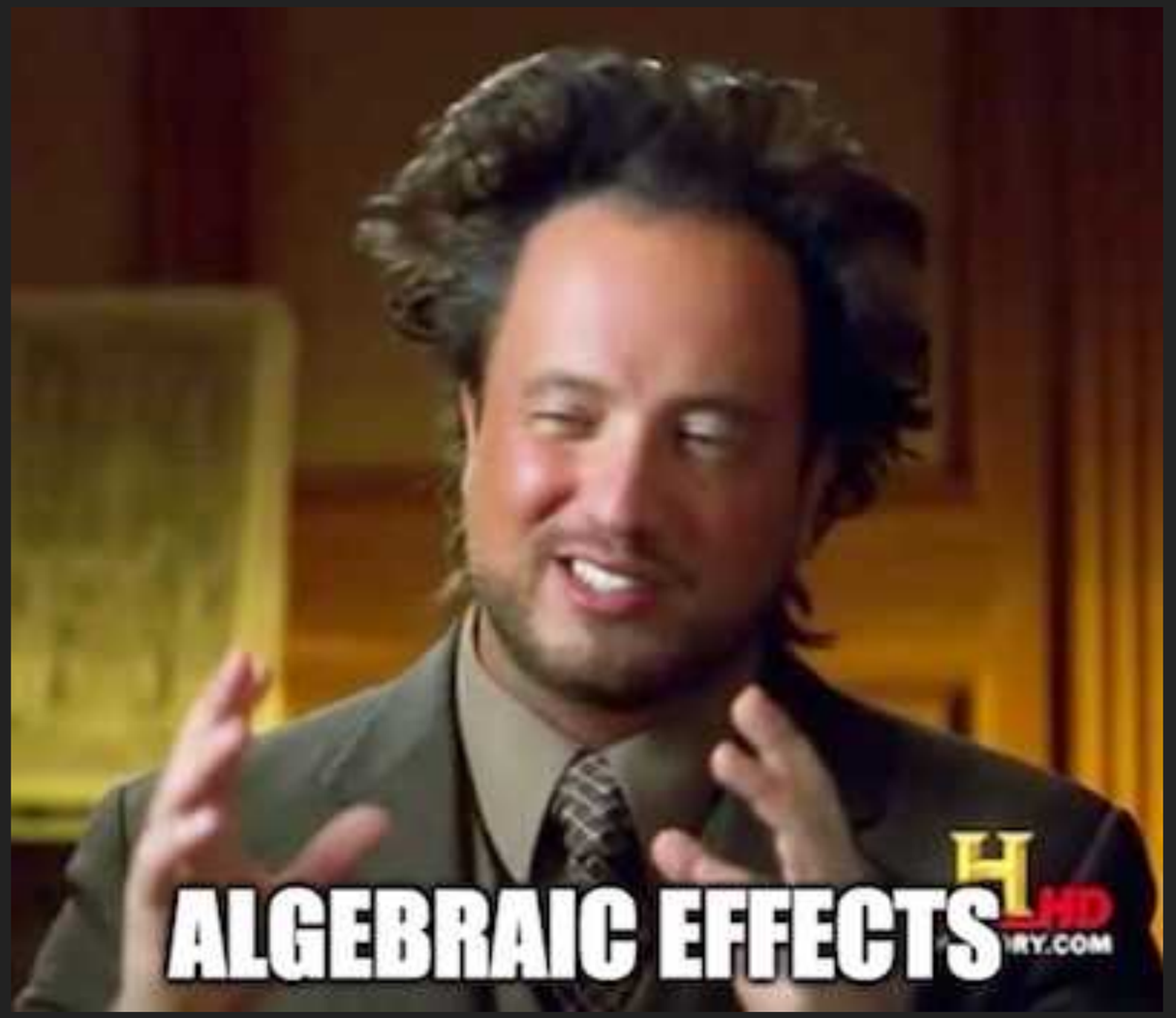
SHONKY

https://github.com/yallop/effects-bibliography

FRANK

UNISON

OCAML


ALGEBRAIC EFFECTS

OVERVIEW

- What are effects?
- But wait … haven’t we known about them for a long while now?
- What is “algebraic” about them?
- Visit Koka
- Visit Ocaml
- Q&A

COMPUTATIONAL EFFECTS

In my opinion, the most useful definition of an “effect” is a computation that alters its environment.

- Sophia Gold on Quora

COMPUTATIONAL EFFECTS

- An effect is most easily understood as an interaction between a sub-expression and a central authority that administers the global resources of a program.
- an effect can be viewed as a message to the central authority plus enough information to resume the suspended calculation.

- Cartwright & Felleisen (1994)

COMPUTATIONAL EFFECTS

- ... examples are various forms of nondeterminism, printing, or jumps of various kinds. These computational effects form the focus of our investigation.
- Computational effects invariably arise from *operations* such as a nondeterministic choice operation, operations for writing or reading, or operations for looking up or updating state.

- Plotkin & Power (2004)

WAIT, BUT . . . .

IMPERATIVE PROGRAMMING

GOTO CONSIDERED HARMFUL

- Djikstra (1968)

LAMBDA – THE ULTIMATE IMPERATIVE

LAMBDA – THE ULTIMATE GOTO

- Sussman, Steele et al. (~1975)

That is, in this *continuation-passing programming style*, a function always “returns” its result by “sending” it to another function. This is the key idea.

CONTINUATION PASSING STYLE

A function written in continuation-passing style takes an extra argument: an explicit "continuation"; *i.e.*, a function of one argument.

When the CPS function has computed its result value, it "returns" it by calling the continuation function with this value as the argument.

That means that when invoking a CPS function, the calling function is required to supply a procedure to be invoked with the subroutine's "return" value.

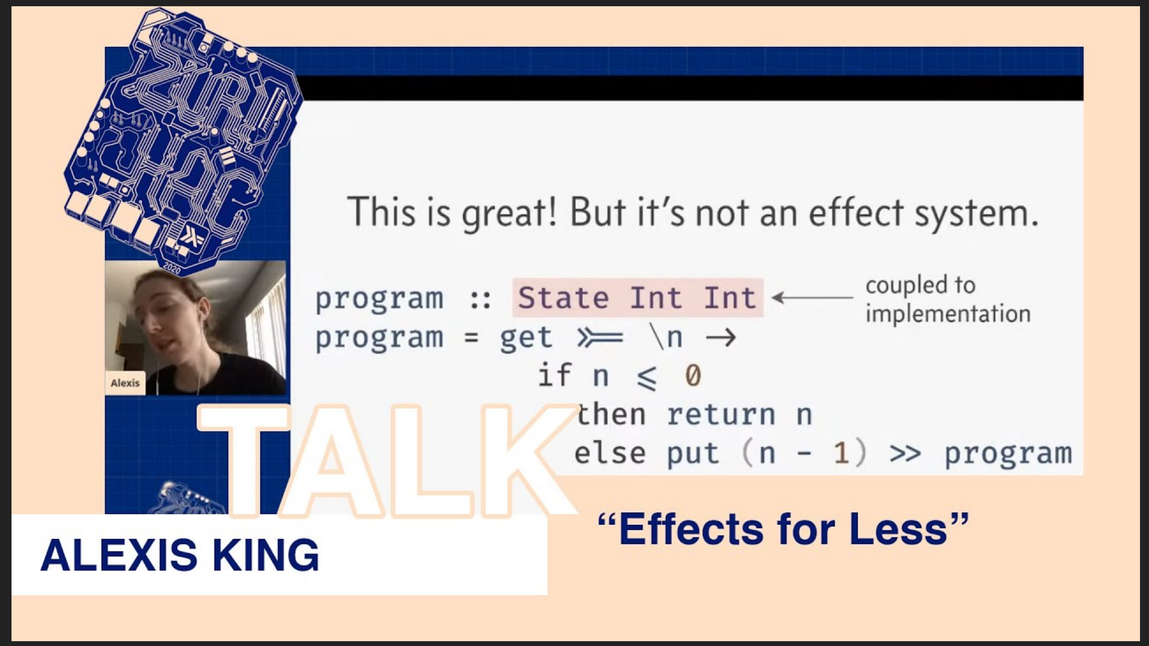
DIRECT VS CONTINUATION-PASSING STYLE

```haskell
pow2 :: Float -> Float
pow2 a = a ** 2

add :: Float -> Float -> Float
add a b = a + b

pyth :: Float -> Float -> Float
pyth a b = sqrt (add (pow2 a) (pow2 b))
```

```haskell
pow2' :: Float -> (Float -> a) -> a
pow2' a cont = cont (a ** 2)

add' :: Float -> Float -> (Float -> a) -> a
add' a b cont = cont (a + b)

-- Types a -> (b -> c) and a -> b -> c are equivalent, so CPS function
-- may be viewed as a higher order function
sqrt' :: Float -> ((Float -> a) -> a)
sqrt' a = \cont -> cont (sqrt a)

pyth' :: Float -> Float -> (Float -> a) -> a
pyth' a b cont = pow2' a (\a2 -> pow2' b (\b2 -> add' a2 b2 (\anb -> sqrt' anb cont)))
```

DELIMITED CONTINUATIONS

- *Delimited continuation* (aka composable continuation or partial continuation), is a “slice” of a continuation frame that has been reified into a function.
- Unlike regular continuations, delimited continuations return a value, and thus may be reused and composed.
- Delimited continuations allow capturing slices of the call stack and restoring them later.



WAIT, BUT . . . .

MONADS

MONADS

- *Computational lambda-calculus and monads*, Moggi 1989
- *Monads provide a convenient framework for simulating effects found in other languages, such as global state, exception handling, output, or non-determinism*, Wadler 1995
- *Programming with monads strongly reminiscent of continuation-passing style (CPS) … In a sense they are equivalent: CPS arises as a special case of a monad, and any monad may be embedded in CPS by changing the answer type*, Wadler *1992*

MONADS, APPLICATIVES, ARROWS

- Monopolized modeling computational effects (especially in pure functional languages)
- Newer classes of effectful computations were introduced via Applicative Functors (McBride & Patterson, 2008) and Arrows (Hughes, 2000)
- Monads do not compose in general

MONAD TRANSFORMERS

- Monad transformers add functionality of one monad to another. They do so by stacking a transformer version of the additional monad on top of the original one, which with enough repetition results in something like a monadic Voltron.

  - Sophia Gold

OK, SO ....

ALGEBRAIC EFFECTS

ALGEBRAIC EFFECTS

- *Algebraic Operations and Generic Effects,* Plotkin & Power (Applied categorical structures, 2003)
- *Handlers of algebraic effects,* Plotkin & Pretnar (2009)
- *Programming with algebraic effects and handlers,* Brauer & Pretnar (2015)
- *Concurrent System Programming with Effect Handlers,* Dolan, KC, Madhavapeddy, et al. (2018)
- *Retrofitting Effect Handlers into OCaml,* KC, Dolan, Madhavapeddy, et al. (2021)

ALGEBRAIC EFFECTS

In a separate development, algebraic effects and handlers (Plotkin & Pretnar, 2009) were created as a more convenient formulation of monadic effects and programs.

Their success is largely due to their **easier integration with impure functional and imperative languages** to enable user-defined effects. This approach encodes effects as operations represented by the signature of an algebraic theory. The semantics of these effects is represented by an interpretation for the operations.

- *Generalized monoidal effects and handlers,* Pieters, Rivas, Schrijvers 2020

ALGEBRAIC EFFECTS

- Computational effects are modelled as operations of a suitably chosen algebraic theory. Common computational effects such as input, output, state, exceptions, and non-determinism, are of this kind

- Effect handlers are a related notion which encompasses exception handlers, stream redirection, transactions, backtracking, and many others. These are modelled as homomorphisms induced by the universal property of free algebras.

  - *Programming with Algebraic Effects and Handlers,* Bauer & Pretnar 2015

Shall I be pure or impure?, Wadler 1992

We say “Yes”: purity is a choice to make locally.

We introduce Frank, an applicative language where the meaning of “impure” computations is open to negotiation, based on Plotkin and Pretnar’s effect handlers (Plotkin & Pretnar, 2013)—a rich foundation for effectful programming.

By separating effect interfaces from their implementation, effect handlers offer a high degree of modularity. Programmers can express effectful programs independently of the concrete interpretation of their effects. A handler gives one interpretation of the effects of a computation.

In Frank, effect types (sometimes called simply effects in the literature) are known as abilities. An ability denotes the permission to invoke a particular set of commands.

- *Doo bee doo bee doo*, Convent, Lindley, McBride, McLaughlin (2020)

ALGEBRAIC HANDLERS

- Effect handlers are a mechanism for modular programming with user-defined effects.
- Effect handlers allow the programmers to describe computations that perform effectful operations, whose meaning is described by handlers that enclose the computations.
- **Effect handlers are a generalization of exception handlers and enable non-local control-flow mechanisms such as resumable exceptions, lightweight threads, coroutines, generators and asynchronous I/O to be composably expressed.**

KOKA

KOKA: A FUNCTIONAL LANGUAGE WITH EFFECTS

- Koka is a strongly typed functional-style language with effect types and handlers.
- Koka tracks the (side) effects of every function in its type, where pure and effectful computations are distinguished.
- Effect handlers let you define advanced control abstractions, like exceptions, async/await, or probabilistic programs, as a user library in a typed and composable way.

Koka documentation

EFFECT TYPING

```
fun sqr    : (int) -> total int        // total: mathematical total function
fun divide : (int,int) -> exn int      // exn: may raise an exception (partial)
fun turing : (tape) -> div int         // div: may not terminate (diverge)
fun print  : (string) -> console ()    // console: may write to the console
fun rand   : () -> ndet int            // ndet: non-deterministic
```

POLYMORPHIC EFFECT TYPES

Consider mapping a function over a list

```
fun map( xs : list<a>, f : a -> e b ) : e list<b>
  match xs
    Cons(x,xx) -> Cons( f(x), map(xx,f) )
    Nil        -> Nil
```

EFFECT HANDLERS

```
effect yield
  ctl yield( i : int ) : bool
```

<- example of an effect definition with one control (ctl) operation to yield int values

```
fun traverse( xs : list<int> ) : yield ()
  match xs
    Cons(x,xx) -> if yield(x) then traverse(xx) else ()
    Nil        -> ()
```

<- use our effect type

The traverse function calls yield and therefore gets the yield effect type.

To use traverse we need to *handle* the yield effect.

```
fun print-elems() : console ()
  with ctl yield(i)
    println("yielded " ++ i.show)
    resume(i<=2)
  traverse([1,2,3,4])
```

The with statement binds the handler for `yield` control operation over the rest of the scope, in this case `traverse([1,2,3,4])`. Every time `yield` is called, our control handler is called, prints the current value, and resumes to the call site with a boolean result.

Note how the handler discharges the yield effect – and replaces it with a console effect.

OCAML

Note: The material in the following slides has been extracted from the from the chapter on Effect Handlers in the (yet to be published) OCaml manual (v5.0).

```
open Effect
open Effect.Deep

type _ Effect.t += Xchg: int -> int t
let comp1 () = perform (Xchg 0) + perform (Xchg 1)
```

define an effect (that is, an operation) that takes an integer argument and returns an integer result. We name this effect Xchg.

We declare the exchange effect Xchg by extending the pre-defined extensible variant type `Effect.t` with a new constructor `Xchg: int -> int t`.

The declaration may be intuitively read as *the Xchg effect takes an integer parameter, and when this effect is performed, it returns an integer*. The computation `comp1` performs the effect twice using the `perform` **primitive** and returns their sum.

OCAML

```
try_with comp1 ()
{ effc = fun (type a) (eff: a t) ->
    match eff with
    | Xchg n -> Some (fun (k: (a, _) continuation) ->
        continue k (n+1))
    | _ -> None }
- : int = 3
```

We can handle the Xchg effect by implementing a handler that always returns the successor of the offered value

`try_with` runs the computation `comp1 ()` under an effect handler that handles the Xchg effect. As mentioned earlier, effect handlers are a generalization of exception handlers. Similar to exception handlers, when the computation performs the Xchg effect, the control jumps to the corresponding handler. However, unlike exception handlers, the handler is also provided with the delimited continuation k, which represents the suspended computation between the point of perform and this handler.

The expressive power of effect handlers comes from the delimited continuation.

While the previous example immediately resumed the computation, the computation may be resumed later, running some other computation in the interim. Let us extend the previous example and implement message-passing concurrency between two concurrent computations using the Xchg effect. We call these concurrent computations *tasks*.

```
type 'a status =
  Complete of 'a
| Suspended of {msg: int; cont: (int, 'a status) continuation}
```

A task either is complete, with a result of type `'a`, or is suspended with the message `msg` to send and the continuation cont. The type `(int,'a status) continuation` says that the suspended computation expects an `int` value to resume and returns a `'a status` value when resumed.

Next, we define a step function that executes one step of computation until it completes or suspends:

```
let step (f : unit -> 'a) () : 'a status =
  match_with f ()
  { retc = (fun v -> Complete v);
    exnc = raise;
    effc = fun (type a) (eff: a t) ->
      match eff with
      | Xchg msg -> Some (fun (cont: (a, _) continuation) ->
            Suspended {msg; cont})
      | _ -> None }
```

The argument to the `step` function, `f`, is a computation that can perform an `Xchg` effect and returns a result of type `'a`. The step function itself returns a `'a status` value.

We can now write a simple scheduler that runs a pair of tasks to completion:

```
let rec run_both a b =
  match a (), b () with
  | Complete va, Complete vb -> (va, vb)
  | Suspended {msg = m1; cont = k1},
    Suspended {msg = m2; cont = k2} ->
      run_both (fun () -> continue k1 m2)
               (fun () -> continue k2 m1)
   | _ -> failwith "Improper synchronization"
```

Both of the tasks may run to completion, or both may offer to exchange a message. In the latter case, each computation receives the value offered by the other computation. The situation where one computation offers an exchange while the other computation terminates is regarded as a programmer error, and causes the handler to raise an exception

We can now define a second computation that also exchanges two messages:

```
let comp2 () = perform (Xchg 21) * perform (Xchg 21)
```

Finally we can run the two computations together:

```
run_both (step comp1) (step comp2)
- : int * int = (42, 0)
```

The computation `comp1` offers the values `0` and `1` and in exchange receives the values `21` and `21`, which it adds, producing `42`. The computation `comp2` offers the values `21` and `21` and in exchange receives the values `0` and `1`, which it multiplies, producing `0`. The communication between the two computations is programmed entirely inside `run_both`. Indeed, the definitions of `comp1` and `comp2`, alone, do not assign any meaning to the `Xchg` effect.

EIO — EFFECTS IN DIRECT STYLE

- The `Unix` library provided with OCaml uses blocking IO operations, and is not well suited to concurrent programs such as network services or interactive applications. For many years, the solution to this has been libraries such as `Lwt` and `Async`, which provide a monadic interface. These libraries allow writing code as if there were multiple threads of execution, each with their own stack, but the stacks are simulated using the heap.
- OCaml 5.0 adds support for "effects", removing the need for monadic code here. Using effects brings several advantages
- Due to this, we anticipate many OCaml users will want to rewrite their IO code once OCaml 5.0 is released. It would be very beneficial to use this opportunity to standardise a single concurrency API for OCaml, and we hope that Eio will be that API.

WRAPPING UP

- Algebraic Effects are a mathematical formalism for effect systems.
- Effect handlers are a generalization of exception handlers and enable non-local control-flow mechanisms such as resumable exceptions, lightweight threads, coroutines, generators and asynchronous I/O to be composably expressed.
- Using the Plotkin & Pretnar machinery to write programs involves making peace with (delimited)-continuation-passing-style and the aesthetics of this approach.
- OCaml 5 has adopted this but also provided machinery for using concurrency primitives via a direct style.
- I am curious to know more about where these systems really come into their own. There's a lot more to learn 🧐.

REFERENCES ❤️

- Know what your functions are doing? - Side effects in 12+ languages (YouTube)
- What is algebraic about algebraic effects and handlers? (PDF and videos)
- An introduction to algebraic effects and handlers (PDF)
- Alexis King - Effects for Less @ ZuriHac 2020 (YouTube)
- KC Sivaramakrishnan - OCaml 5.0 - Concurrent and Parallel programming @ICFP 2022 (Slides)
- KC Sivaramakrishnan - Retrofitting Concurrency - Lessons from the Engine Room @ ICFP 2022 (YouTube)