Effective programming
Bringing algebraic effects and handlers to OCaml

Leo White\textsuperscript{1}  Stephen Dolan\textsuperscript{2}  Matija Pretnar\textsuperscript{3}  KC Sivaramakrishnan\textsuperscript{2}

\textsuperscript{1}Jane Street
\textsuperscript{2}University of Cambridge  \textsuperscript{3}University of Ljubljana
Algebraic effects and handlers
Algebraic effects and handlers

- Algebraic effects originally introduced to study the semantics of computational effects.
  - *Algebraic Operations and Generic Effects*
    - Plotkin and Power, 2002
Algebraic effects and handlers

- Algebraic effects originally introduced to study the semantics of computational effects.
  - *Algebraic Operations and Generic Effects*
    Plotkin and Power, 2002

- The addition of handlers turned them into a construct for implementing such effects.
  - *Handlers of Algebraic Effects*
    Plotkin and Pretnar, 2009
Simple example

```ml
let f () =
  (perform Get) + (perform Get) + 2

match f () with
| ret -> ret
| effect Get, k -> continue k 9

:- int = 20

match f () with
| ret -> ret
| effect Get, k -> continue k 99

:- int = 200
```
Syntax

Performing effects

\[ e ::= \ldots \mid \text{perform } E \ e^? \]

Handling effects

\[ e ::= \ldots \]

\[ \quad \mid \text{match } e \text{ with } \]

\[ \quad \quad (\mid x \rightarrow e)^* \]

\[ \quad \quad (\mid \text{effect } E \ x^?, \ x \rightarrow e)^* \]

Resuming a continuation

\[ e ::= \ldots \mid \text{continue } e \ e \]
Typing (unchecked)

\[ E : A \to B \quad \Gamma \vdash e : A \]
\[ \Gamma \vdash \text{perform } E \ e : B \]

\[ \Gamma \vdash e : (A,B) \text{cont} \quad \Gamma \vdash e' : A \]
\[ \Gamma \vdash \text{continue } e \ e' : B \]
Typing (unchecked)

\[
\begin{align*}
\Gamma & \vdash e : A & E_i & : C_i \rightarrow D_i \\
\Gamma ; x : A & \vdash e' : B & \Gamma ; x_i : C_i ; k_i : (D_i, B) \text{ cont} & \vdash e''_i : B
\end{align*}
\]

match e with

\[
\begin{align*}
\Gamma & \vdash \mid x \rightarrow e' : B \\
\mid \text{effect } E_i \ x_i , k_i & \rightarrow e''_i
\end{align*}
\]
Semantics

\[ v ::= \ldots \text{(values)} \]
\[ r ::= v \mid \text{effect } E v v \text{(results)} \]
\[ C[\_] ::= \ldots \text{(delimited contexts)} \]

\[ C[\text{perform } E v] \longrightarrow \text{effect } E v (\lambda x. C[x]) \]
\[ \text{continue } v v' \longrightarrow v v' \]
match $v$ with
t $\rightarrow e$ $\quad \rightarrow e[v/x]$ 
effect $E_i \; x_i, \; k_i \rightarrow e'_i$
Semantics

```
match effect E v v' with
| x -> e                      → e'_j[v/x, v_{cont}/k]
| effect E_i x_i, k_i -> e'_i

where E = E_j and

match v'y with
v_{cont} = \lambda y. | x -> e
| effect E_i x_i, k_i -> e'_i
```
Semantics

\[
\begin{align*}
C &\left[\text{match } E \nu \nu' \text{ with} \right. \\
&\quad | x \to e' \\
&\quad | \text{effect } E_i \ x_i \ , \ k_i \to e''_i \\
\left. \right] \\
\rightarrow \\
\text{match } \nu' \ y \text{ with} \\
\text{effect } E \nu (\lambda y. C [x \to e' \\
&\quad | \text{effect } E_i \ x_i \ , \ k_i \to e''_i ])
\end{align*}
\]

where \( \forall j. E \neq E_j \)
Examples: Exceptions

```ml
let raise (msg : string) : 'a = perform Raise msg

let run f =
    match f () with
    | ret -> Ok ret
    | effect Raise msg, k -> Error msg
```
Examples: State

```plaintext
let put (v : int) : unit = perform Put v

let get () : int = perform Get

let run init f =
  let comp =
    match f () with
    | ret ->
      (fun s -> ret)
    | effect Put s', k ->
      (fun s -> continue k () s')
    | effect Get, k ->
      (fun s -> continue k s s)
  in
  comp init
```
Examples: Choice

```ocaml
let select () : bool = perform Select

let run_true f =
  match f () with
  | ret -> ret
  | effect Select, k ->
      continue k true

let run_all f =
  match f () with
  | ret -> [ret]
  | effect Select, k ->
      continue k true @ continue k false
```
Algebraic effects in OCaml
Defining (unchecked) effects

```effect
Get : int
```

```effect
Put : int -> unit
```
Default handlers

effect Yield : unit
   with function Yield -> ()
Affine continuations

let select () : bool = perform Select

let run_all f =
  match f () with
  | ret -> [ret]
  | effect Select, k ->
    continue k true @ continue k false

let _ = run_all select

Exception: Invalid_argument "continuation already taken"
Implementation

- Fibers: Heap allocated, dynamically resized stacks
  - 10s of bytes

- Entering an effect handler creates a fresh fiber

- Call stack becomes a linked list of fibers
Implementation

handler

handle / continue

sp

call chain
reference
Implementation
Implementation
Fibers

- Stack overflow checks for OCaml functions
- Simple static analysis eliminates many checks
- FFI calls are more expensive due to stack switching
Normalized time (lower is better)

Fibers around 0.9% slower
Algebraic effects for concurrency
Concurrency effects

effect Async : ('a -> 'b) * 'a -> 'b promise

effect Await : 'a promise -> 'a

effect Write :
  file_descr * bytes * int * int -> int
with function Write(fd, buf, ofs, len) ->
  Unix.write fd buf ofs len
...

let rec schedule state =
  if Queue.is_empty state.run_q then
    if empty state.reads &&
      empty state.writes then ()
    else select state
  else
    Queue.pop state.run_q ()
let wait state p k =  
  match !p with  
  | Done v -> continue k v  
  | Waiting l ->  
    p := Waiting (k::l);  
    schedule state

let finish state p v =  
  match !p with  
  | Waiting l ->  
    p := Done v;  
    List.iter (fun k ->  
      Queue.push (fun () -> continue k v)  
      state.run_q)  
    l  
  | _ -> assert false
Scheduler

```ml
let rec run state p f x ->
    match f x with
    | v -> finish state p v; schedule state
    | effect Async(f, x), k ->
        let p = promise () in
        Queue.push (fun () -> continue k p)
        state.run_q;
        run state p f x
    | effect Await p, k -> wait state p k
```
val async : ('a -> 'b) -> 'a -> 'b future
val await : 'a future -> 'a

val write :
    file_descr -> bytes -> int -> int -> int ...

val run : (unit -> unit) -> unit
An effect system for OCaml
Effect system

\[ A, B, \ldots ::= \ldots | A \xrightarrow{\Delta} B \]

\[
\frac{\Gamma ; x : A \vdash e : B ! \Delta}{\Gamma \vdash \lambda x. e : A \xrightarrow{\Delta} B ! []} \\
\frac{\Gamma \vdash e : A \xrightarrow{\Delta} B ! \Delta \quad \Gamma \vdash e' : A ! \Delta}{\Gamma \vdash e \; e' : B ! \Delta}
\]
Requirements

Soundness
If a program receives a type $A ! \Delta$, every potential effect $e$ should be captured in $\Delta$.

Usefulness
An effect system that annotates each program with every possible effect there is, is obviously sound, but not very useful. Thus, an effect information should not mention an effect that is guaranteed not to happen.

Backwards compatibility
We want each program that was typable before introducing effects to remain typable.
if $e$ then perform $E_1$
else perform $E_2$
if e then perform $E_1$
else perform $E_2$

Two established approaches to providing the required flexibility:

- Subtyping
- Row polymorphism
Subtyping

\[ \Gamma \vdash e : A ! \Delta \quad A <: B \]
\[ \Gamma \vdash e : B ! \Delta \]

Full implicit subtyping is difficult to add to OCaml:

- OCaml supports invariant type parameters.
- Requires \textit{constrained types} of the form \( A \mid C \) where \( C \) is a set of constraints between type parameters.
- Constrained types do not interact well with OCaml’s module system.
- Constraint generation needs to be directed to correctly track variance.
Row polymorphism

$$\Delta ::= [\varepsilon | \Delta] \mid [\rho] \mid [\ ]$$

$$\Delta \equiv \Delta'$$

$$[\varepsilon | \Delta] \equiv [\varepsilon | \Delta']$$

$$[\varepsilon | \varepsilon' | \Delta] \equiv [\varepsilon' | \varepsilon | \Delta]$$
Row polymorphism

\[ E : A \rightarrow B \in \mathcal{E} \quad \Gamma \vdash e : A ! [\mathcal{E} \mid \Delta] \]

\[ \Gamma \vdash \text{perform } E \ e : B ! [\mathcal{E} \mid \Delta] \]
Row polymorphism

```ocaml
let raise msg = perform Raise msg ;;

val raise : string -[exn | !p]--> unit
```
Row polymorphism

\[
\begin{align*}
\Gamma \vdash e : A ! [\mathcal{E} | \Delta] & \quad \mathcal{E} = \{ E_i : C_i \rightarrow D_i \} \\
\Gamma ; x : A \vdash e' : B ! \Delta & \quad \Gamma ; x_i : C_i \; k_i : (D_i, B) \text{ cont} \vdash e''_i : B ! \Delta \\
\text{match } e \text{ with} & \\
\Gamma \vdash | x \rightarrow e' : B ! \Delta \\
| \text{effect } E_i \; x_i, k_i \rightarrow e''_i
\end{align*}
\]
let run f =
  match f () with
  | ret -> Ok ret
  | effect Raise msg, k -> Error msg

val run : (unit -> 'a) -> ('a, string) result
Row polymorphism

```ocaml
val old_fun : int -> int

let new_fun p =
  if p then old_fun 10
  else perform Get

Error: This expression performs effect \([\text{state}]!r\), but it was expected to perform \([\text{io}]\).
```
Row polymorphism

```haskell
type t = int -> int

Error: Unbound type parameter !r.
```
A compromise

\[
\Gamma \vdash e : \forall \alpha \rho. A ! \Delta \quad \text{open}^+(A) = \forall \rho'. B \\
\Gamma \vdash e : B[\overline{C}/\overline{\alpha}, \overline{\Delta'}/\overline{\rho}, \overline{\Delta''}/\overline{\rho'}] ! \Delta
\]

\[\text{open}^+([\mathcal{E}_1 | \ldots | \mathcal{E}_n]) = \forall \rho.[\mathcal{E}_1 | \ldots | \mathcal{E}_n | \rho]\]

\[\text{open}^+(A \xrightarrow{\Delta} B) = \text{open}^-(A) \xrightarrow{\text{open}^+ \Delta} \text{open}^+(B)\]

\[\ldots\]

\[\text{open}^-([\mathcal{E}_1 | \ldots | \mathcal{E}_n]) = [\mathcal{E}_1 | \ldots | \mathcal{E}_n]\]

\[\text{open}^-(A \xrightarrow{\Delta} B) = \text{open}^+(A) \xrightarrow{\text{open}^- \Delta} \text{open}^-(B)\]

\[\ldots\]
A compromise

```ocaml
val old_fun : int -> int

let new_fun p =
  if p then old_fun 10
  else perform Get

val new_fun : bool - [state | !p] -> int```

A compromise

\[
\Gamma \vdash e : A ! [] \\
\alpha \rho \not\in ftv(\Gamma) \\
\text{close}^+(\forall \alpha \rho . A) = \forall \alpha \rho'. B \\
\Gamma \vdash e : \forall \alpha \rho'. B ! \Delta
\]

\[
\text{close}^+(\forall \alpha \rho . A) = \forall \alpha \rho . A[[] / \text{closable}^+(A, \rho)]
\]

closable\(^+\)(\Delta, \rho) = \rho

closable\(^+\)(A \xrightarrow{\Delta} B, \rho) = \text{closable}^-\(A, \rho\) \cap \text{closable}^+\(\Delta \cap \text{closable}^+(B)\)

\[
\text{closable}^-\([\mathcal{E}_1 | \ldots | \mathcal{E}_n | \rho], \rho\) = \rho \setminus \rho
\]

closable\(^-\)(A \xrightarrow{\Delta} B, \rho) = \text{closable}^+(A, \rho) \cap \text{closable}^-\(\Delta \cap \text{closable}^-\(B)\)

\[
\ldots
\]
A compromise

```ocaml
let raise msg = perform Raise msg ;;
val raise : unit -[exn]-> int
```
Defining effects

```
effect state =
  | Get : int
  | Put : int -> unit
```
Defining effects

effect fail =
  | Failure of string
Define a built-in abstract effect:

```ocaml
effect io
```

Treat OCaml’s built-in side-effects as performing it:

```ocaml
val ref : 'a -[io]-> 'a ref
```

As with Haskell, divergence and raising exceptions are still considered “pure”.
Usability
Useful short-hands

->    =    -[io]->

->>>  =    -[[]]->

~>    =    -[io | !~]->

~>>>  =    -[!~]->
Useful short-hands

val map : ('a ->> 'b) ->>> 'a list ~>> 'b list
val map : ('a ~> 'b) ->>> 'a array ~> 'b array
Updating the standard library

- The standard library is 101 files totalling 23675 lines
- 72 files changed, 3 insertions(+), 160 deletions(-), 4618 modifications(!)
- 2410 lines: changing value specifications – no explicit effect variables needed

- `val map : ('a -> 'b) -> 'a list -> 'b list`
  + `val map : ('a ~>> 'b) -> 'a list ~>> 'b list`

- 220 lines: avoiding polymorphic comparison

- `if x = y then`
  + `if Int_compare.(x = y) then`
Updating the standard library

- 214 lines: pure versions of Set and Map – implementations shared with impure versions but some boilerplate required

```ocaml
+module type OrderedTypePure =
+  sig
+    type t
+    val compare : t -> t -> int
+  end
```

- 1892 lines: Adding an effect parameter to format strings.

```ocaml
-val val printf :
   ('a, out_channel, unit) format -> 'a
+val val printf :
   ('a, out_channel, unit, ![io | !p]) format
   -> ![io | !p] -> 'a
```
Updating the standard library

- And 2 type annotations:

  - let printers = ref []
  + let printers :
      (exn -> string option) list ref = ref []

  - let locfmt = format_of_string "...";;
  + let locfmt : _ format6e = "...";;
Replacing Not_found the standard library

- 34 files changed, 31 insertions(+), 332 modifications(!)

- 130 lines changing raise to perform and with to with effect

-raise Not_found
+perform Not_found

- 158 lines: updating value specifications

-val find :
    (’a ~>> bool) ~>> ’a list ~>> ’a
+val find :
    (’a ~-[not_found | !p]~>> bool) ~>>
    ’a list ~-[not_found | !p]~>> ’a
Replacing `Not_found` in the standard library

- 35 lines: adding handlers for cases that were not expected to occur

```ocaml
+try
  min_binding t
+with effect Not_found -> assert false
```

- 1 type annotation

```ocaml
-and parse_integer str_ind end_ind =
+and parse_integer :
  int ->> int -> int * int =
```
Replacing Not\_found in the standard library

- 2 coercions related to sharing implementations between the pure and impure versions of Set/Map

```ocaml
let compare_not_found =
+ (Ord.compare
+  : _ -[.. as ![] E.eff]-> _
+  -[.. as ![] E.eff]-> _
+  :> _ -[not\_found | .. as ![] E.eff]-> _
+  -[not\_found | .. as ![] E.eff]-> _)
```
Typed concurrency effects

effect async =
  | Async :
  (’a -> [aio|async|io] -> ’b) * ’a -> ’b promise
  | Await : ’a promise -> ’a

effect aio =
  | Write :
    file_descr * bytes * int * int -> int
  | ...
  with function
    | Write(fd, buf, ofs, len) ->
      Unix.write fd buf ofs len
    | ...
Typed concurrency interface

effect async

val async :
  ('a -[async|aio|io]-> 'b) ->> 'a
  -[async]-> 'b promise
val await : 'a promise -[async]-> 'a

effect aio with function

val write :
  file descr ->> bytes ->> int ->> int
  -[aio]-> int

val run : (unit -[async|aio|io]-> unit) -> unit
Challenges
effect yield = Yield : unit

let f () =
    match perform Yield with
    | _ -> 'None
    | effect Yield, k -> 'Some k

let x = f ()
let y = f ()

let _ =
    match x, y with
    | 'Some x, 'Some y ->
        continue x (), continue y ()
    | p -> p
Affine continuations and purity

effect yield = Yield : unit

let f () =
    match perform Yield with
    | _ -> 'None
    | effect Yield, k -> 'Some k

let x = f ()
let y = x

let _ =
    match x, y with
    | 'Some x, 'Some y ->
        continue x (), continue y ()
    | p -> p
Effect parameters and abstraction

```ocaml
let fold f l init =
  let comp =
    match
      List.iter
        (fun x ->
          perform Put (f x (perform Get))) l
    with
    | () -> fun s -> s
    | effect Get, k -> fun s -> continue k s
    | effect Put s, k ->
        fun _ -> continue k () s
  in
  comp init
```
```
Effect parameters and abstraction

```ml
let fold (type acc) f l init = 
  let effect state = 
   | Get : acc 
   | Put : acc -> unit in 
  let comp = 
    match 
      List.iter 
        (fun x -> 
          perform Put (f x (perform Get))) l 
    with 
      | () -> fun s -> s 
      | effect Get, k -> fun s -> continue k s s 
      | effect Put s, k -> 
        fun _ -> continue k () s 
    in 
  comp init
```
Effect parameters and abstraction

```plaintext
module M : sig
  effect 'a fold2
  val pfrm : unit -> int fold2 -> unit
  val handle : ('a -> int fold2 | !p) -> 'b ->
    'a -> !p -> 'b
end = struct
  effect 'a fold2 = 'a fold
  let pfrm () = perform Put 0
  let handle f x =
    match f x with
    | y -> y
    | effect Get, k = continue k 0
    | effect Put _, k = continue k ()
end
```
Effect parameters and abstraction

```ml
let _ =
    M.handle (fun () ->
        let comp =
            match M.pfrm (); perform Get with
            | x -> fun _ -> x
            | effect Get, k ->
                fun s -> continue k s s
            | effect Put s, k ->
                fun _ -> continue k () s
        in
    print_string (comp "init")
```
Nominative vs Structural

- Nominative definitions in OCaml are all abstractable. Can’t really restrict abstraction whilst effects are treated nominatively.

- Could avoid abstraction by treating effects structurally:
  ```ocaml
  let get : unit -> 'a = fun () -> perform 'Get
  ```

- Allows parameterised effects

- How to handle io – which is an abstract effect?

- How to handle default handlers?
So...

- Algebraic effects and handlers are a good mechanism for modelling effects
- Algebraic effects and handlers enable users to efficiently and composably implement their own concurrent schedulers
- Effect systems can be used to manage algebraic effects as well as side-effects more generally
- It is possible to create effect systems that are both usable and backwards compatible with existing languages like OCaml