Effective programming

Bringing algebraic effects and handlers to OCaml

Leo White[1] Stephen Dolan[2] Matija Pretnar[3] KC Sivaramakrishnan[2]

[1] Jane Street

[2] University of Cambridge [3] University of Ljubljana

Algebraic effects and handlers

Algebraic effects and handlers

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

Algebraic effects and handlers

- Algebraic effecects 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

```
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 ::= $\ldots$ | `perform` E $e^?$

Handling effects

e ::= $\ldots$
| `match` e `with`
(`|` x `->` e)*
(`|` `effect` E $x^?$`,` x `->` e)*

Resuming a continuation

e ::= $\ldots$ | `continue` e e

Typing (unchecked)

$$\frac{E : A \to B \qquad \Gamma \vdash e : A}{\Gamma \vdash \texttt{perform}\ E\ e : B}$$

$$\frac{\Gamma \vdash e : (A, B)\ \texttt{cont} \qquad \Gamma \vdash e' : A}{\Gamma \vdash \texttt{continue}\ e\ e' : B}$$

Typing (unchecked)

$$\frac{\Gamma \vdash e : A \qquad \Gamma ; x : A \vdash e' : B \qquad \begin{array}{c} E_i : C_i \to D_i \\ \Gamma ; x_i : C_i ; k_i : (D_i , B)\ \mathtt{cont} \vdash e''_i : B \end{array}}{\Gamma \vdash \begin{array}{l} \mathtt{match}\ e\ \mathtt{with} \\ \mid x \;\mathtt{->}\; e' \\ \mid \mathtt{effect}\ E_i\ x_i ,\ k_i \;\mathtt{->}\; e''_i \end{array} : B}$$

Semantics

v `::=` ... (values)
r `::=` v | `effect` E v v (results)
$\mathcal{C}$`[_]` `::=` ... (delimited contexts)

$$\mathcal{C}[\texttt{perform}\ E\ v] \longrightarrow \texttt{effect}\ E\ v\ (\lambda x.\mathcal{C}[x])$$

$$\texttt{continue}\ v\ v' \longrightarrow v\ v'$$

Semantics

$$
\begin{array}{ll}
\texttt{match}\ v\ \texttt{with} & \\
\mid x\ \texttt{->}\ e & \longrightarrow e[v/x] \\
\mid \texttt{effect}\ E_i\ x_i\ ,\ k_i\ \texttt{->}\ e'_i &
\end{array}
$$

Semantics

$$\begin{array}{ll}\texttt{match effect}\ E\ v\ v'\ \texttt{with} & \\ \texttt{|}\ x\ \texttt{->}\ e & \longrightarrow e'_j[v/x_j, v_{cont}/k_j] \\ \texttt{|}\ \texttt{effect}\ E_i\ x_i\texttt{,}\ k_i\ \texttt{->}\ e'_i & \end{array}$$

where $E = E_j$ and

$$v_{cont} = \lambda y. \begin{array}{l}\texttt{match}\ v' y\ \texttt{with} \\ \texttt{|}\ x\ \texttt{->}\ e \\ \texttt{|}\ \texttt{effect}\ E_i\ x_i\texttt{,}\ k_i\ \texttt{->}\ e'_i\end{array}$$

Semantics

$$\mathcal{C}\left[\begin{array}{l}\texttt{match effect } E\ v\ v'\ \texttt{with} \\ \mid x\ \texttt{->}\ e' \\ \mid \texttt{effect}\ E_i\ x_i\,,\ k_i\ \texttt{->}\ e''_i\end{array}\right]$$

$$\longrightarrow$$

$$\texttt{effect}\ E\ v\ (\lambda y.\mathcal{C}\left[\begin{array}{l}\texttt{match}\ v'\ y\ \texttt{with} \\ \mid x\ \texttt{->}\ e' \\ \mid \texttt{effect}\ E_i\ x_i\,,\ k_i\ \texttt{->}\ e''_i\end{array}\right])$$

where $\forall j.E \neq E_j$

Examples: Exceptions

```
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

```
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

```
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 -> ()
```

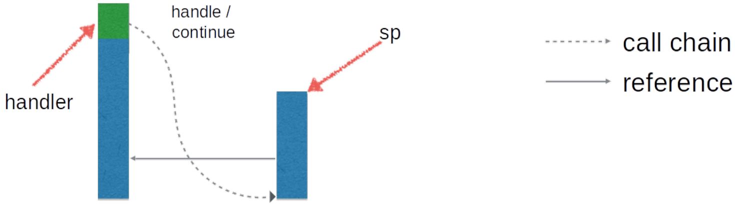
Affine continuations

```
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"
```

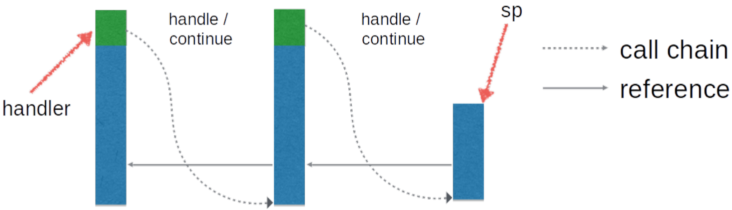
Implementation

- 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

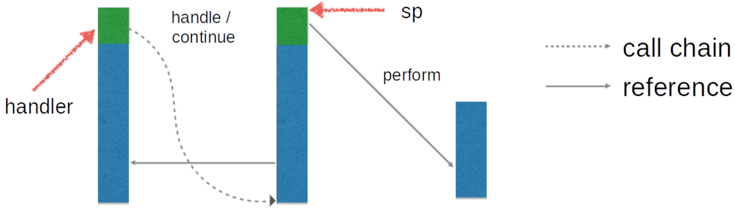
Implementation



Implementation



Implementation

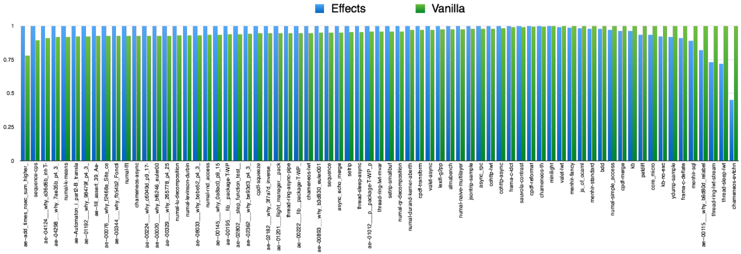


Fibers

- Stack overflow checks for OCaml functions
- Simple static analysis eliminates many checks
- FFI calls are more expensive due to stack switching

Fibers

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
...
```

Scheduler

```
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 ()
```

Scheduler

```
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

```
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
```

Interface

```
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

$$A, B, \ldots \; ::= \; \ldots \mid A \xrightarrow{\Delta} B$$

$$\frac{\Gamma ; x : A \vdash e : B \,!\, \Delta}{\Gamma \vdash \lambda x.e : A \xrightarrow{\Delta} B \,!\, []}$$

$$\frac{\Gamma \vdash e : A \xrightarrow{\Delta} B \,!\, \Delta \qquad \Gamma \vdash e' : A \,!\, \Delta}{\Gamma \vdash e\; e' : B \,!\, \Delta}$$

Requirements

Soundness

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

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.

Requirements

```
if e then perform E1
else perform E2
```

Requirements

```
if e then perform E1
else perform E2
```

Two established approaches to providing the required flexibility:

- Subtyping
- Row polymorphism

Subtyping

$$\frac{\Gamma \vdash e : A \mathrel{!} \Delta \qquad A <: B}{\Gamma \vdash e : B \mathrel{!} \Delta}$$

Full implicit subtyping is difficult to add to OCaml:

- OCaml supports invariant type parameters.
- Requires *constrained types* of the form $A|\mathcal{C}$ where $\mathcal{C}$ 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

Δ `::=` `[` $\mathcal{E}$ `|` Δ `]` $\mid$ `[` ρ `]` $\mid$ `[ ]`

$$\frac{\Delta \cong \Delta'}{[\mathcal{E} \mid \Delta] \cong [\mathcal{E} \mid \Delta']}$$

$$[\mathcal{E} \mid \mathcal{E}' \mid \Delta] \cong [\mathcal{E}' \mid \mathcal{E} \mid \Delta]$$

Row polymorphism

$$\frac{E : A \to B \in \mathcal{E} \qquad \Gamma \vdash e : A \,!\, [\mathcal{E} \mid \Delta]}{\Gamma \vdash \texttt{perform}\ E\ e : B \,!\, [\mathcal{E} \mid \Delta]}$$

Row polymorphism

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

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

Row polymorphism

$$\frac{\begin{array}{cc} \Gamma \vdash e : A \,!\, [\mathcal{E} \mid \Delta] & \mathcal{E} = \left\{ E_i : C_i \to D_i \right\} \\ \Gamma \,;\, x : A \vdash e' : B \,!\, \Delta & \Gamma \,;\, x_i : C_i \,;\, k_i : (D_i \,, B)\ \texttt{cont} \vdash e''_i : B \,!\, \Delta \end{array}}{\Gamma \vdash \begin{array}{l} \texttt{match}\ e\ \texttt{with} \\ \texttt{|}\ x\ \texttt{->}\ e' \\ \texttt{|}\ \texttt{effect}\ E_i\ x_i\ \texttt{,}\ k_i\ \texttt{->}\ e''_i \end{array} : B \,!\, \Delta}$$

Row polymorphism

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

val run : (unit -[exn | !p]-> 'a)
            -[!p]-> ('a, string) result
```

Row polymorphism

```
val old_fun : int -> int

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

```
Error: This expression performs effect [state| !r], but
       it was expected to perform [io].
```

Row polymorphism

```
type t = int -> int
```

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

A compromise

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

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

$$open^{+}(A \xrightarrow{\Delta} B) = open^{-}(A) \xrightarrow{open^{+}\Delta} open^{+}(B)$$

$$\ldots$$

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

$$open^{-}(A \xrightarrow{\Delta} B) = open^{+}(A) \xrightarrow{open^{-}\Delta} open^{-}(B)$$

$$\ldots$$

A compromise

```
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

$$\frac{\Gamma \vdash e : A \,!\, [] \qquad \overline{\alpha\rho} \notin \mathit{ftv}(\Gamma) \qquad \mathit{close}^{+}(\forall\overline{\alpha\rho}.A) = \forall\overline{\alpha\rho'}.B}{\Gamma \vdash e : \forall\overline{\alpha\rho'}.B \,!\, \Delta}$$

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

$$\mathit{closable}^{+}(\Delta,\overline{\rho}) = \overline{\rho}$$

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

$$\ldots$$

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

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

$$\ldots$$

A compromise

```
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
```

Purity

Define a built-in abstract effect:

```
effect io
```

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

```
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 explict 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

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

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

```
-val printf :
  ('a, out_channel, unit) format -> 'a
+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]~>> bool) ->>
        'a list ~[not_found]~>> 'a
  ```

Replacing Not_found in the standard library

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

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

- 1 type annotation

```
-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

```
+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]-> '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]-> '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]-> unit) -> unit
```

Challenges

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 = 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

```
effect 'a state =
    | Get : 'a
    | Put : 'a -> unit

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 s
    | effect Put s, k ->
        fun _ -> continue k () s
  in
  comp init
```

Effect parameters and abstraction

```
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

```
module M : sig
  effect 'a fold2
  val pfrm : unit -[int fold2]->> unit
  val handle : ('a ~[int fold2]~> 'b) ->
                    'a ~> '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

```
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:

```
let get : unit -[ `Get : 'a ]->> 'a =
  fun () -> perform `Get
```

- Allows parameterised effects
- Parameterised effects allow "ST" effect – which allows pure functions using affine continuations.
- 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