Control structures, fifth lecture

The practice of effects: from exceptions to effect handlers

Xavier Leroy

2024-02-23

Collège de France, chair of software sciences

xavier.leroy@college-de-france.fr

Exceptions

Exceptions in a functional language

An exception = a value (type `exn`) that describes an exceptional condition (error, lack of a meaningful result, ...).

Expressions:

$$
\begin{array}{rll}
e ::= & cst \mid x \mid \lambda x.\, e \mid e_1\ e_2 & \\
& \mid \texttt{raise}\ \ e & \text{raising an exception} \\
& \mid \texttt{try}\ \ e_1\ \ \texttt{with}\ \ x \to e_2 & \text{handling an exception}
\end{array}
$$

`raise` e stops evaluation and branches to the nearest enclosing `try`...`with`. This expression returns no value.
(As shown by the type $\texttt{raise} : \forall \alpha, \texttt{exn} \to \alpha$.)

Exceptions in a functional language

An exception = a value (type `exn`) that describes an exceptional condition (error, lack of a meaningful result, …).

Expressions:

$$\begin{array}{lll} e ::= cst \mid x \mid \lambda x.\, e \mid e_1\ e_2 & & \\ \quad \mid \texttt{raise}\ e & & \text{raising an exception} \\ \quad \mid \texttt{try}\ e_1\ \texttt{with}\ x \to e_2 & & \text{handling an exception} \end{array}$$

$\texttt{try}\ e_1\ \texttt{with}\ x \to e_2$ evaluates the body e_1.

If e_1 raises no exception, its value is returned as the value of the whole `try`…`with`.

If e_1 raises an exception, the value v of the exception is bound to x and the handler e_2 is evaluated.

Examples of uses of exceptions

Error reporting (for instance, arithmetic overflow):

```
let safe_add x y =
  let z = x + y in
  if (z lxor x) land (z lxor y) < 0 then raise Overflow;
  z

let sum_list l =
  try
    let s = List.fold_left safe_add 0 l in
    printf "Sum is %d\n" s
  with Overflow ->
    printf "Overflow!\n"
```

Examples of uses of exceptions

Early exit from nested recursive calls:

```
let list_product l =
    let exception Zero in
    let rec product = function
      | [] -> 1
      | 0 :: _ -> raise Zero
      | n :: l -> n * product l
    in
    try product l with Zero -> 0
```

Examples of uses of exceptions

Emulating `break` and `continue`:

```
exception Break in
exception Continue in
try
    for i = lo to hi do
        try
            ... raise Break ... raise Continue ...
        with Continue -> ()
    done
with Break -> ()
```

Exceptions that are raised and handled in the same function
$\approx$ multi-level exit (lecture #1) $\approx$ forward `goto`.

Reduction semantics

Two head-reduction rules for `try...with`:

$$\begin{aligned} \texttt{try}\ v\ \texttt{with}\ x \to e &\ \xrightarrow{\varepsilon}\ v \\ \texttt{try}\ D[\texttt{raise}\ v]\ \texttt{with}\ x \to e &\ \xrightarrow{\varepsilon}\ e\{x \leftarrow v\} \end{aligned}$$

Here, D is a context with no `try...with` enclosing the hole:

Reduction contexts:

$$C ::= [\,] \mid C\ e \mid v\ C \mid \texttt{raise}\ C \mid \texttt{try}\ C\ \texttt{with}\ x \to e$$

Exception propagation contexts:

$$D ::= [\,] \mid D\ e \mid v\ D \mid \texttt{raise}\ D$$

(See later: the semantics of effect handlers here.)

Reduction semantics

Consider a program p that is about to raise exception v:

$$p = C[\texttt{raise } v]$$

If the `raise` v is enclosed in a `try...with`, we write p as

$$p = C'[\texttt{try } D[\texttt{raise } v] \texttt{ with } x \to e]$$

and we reduce

$$p \to C'[e\{x \leftarrow v\}]$$

If the `raise` v is not enclosed in any `try...with`, program p is stuck on an uncaught exception.

Exception-returning style (ERS)

An alternative to exceptions: include errors in the return values of functions.

```
type ('a, 'e) result = V of 'a | E of 'e

let safe_add x y : (int, string) result =
  let z = x + y in
  if (z lxor x) land (z lxor y) < 0
  then E "overflow"
  else V z

let rec safe_add_list = function
  | [] -> V 0
  | x :: l ->
      match safe_add_list l with
      | V y -> safe_add x y
      | E e -> E e
```

The ERS transformation

$$
\begin{aligned}
\mathcal{E}(cst) &= V\ cst \\
\mathcal{E}(x) &= V\ x \\
\mathcal{E}(\lambda x.\, e) &= V\ (\lambda x.\, \mathcal{E}(e)) \\
\mathcal{E}(e_1\ e_2) &= \texttt{match}\ \mathcal{E}(e_1)\ \texttt{with}\ E\,x_1 \to E\,x_1 \mid V\,v_1 \to \\
&\ \texttt{match}\ \mathcal{E}(e_2)\ \texttt{with}\ E\,x_2 \to E\,x_2 \mid V\,v_2 \to v_1\,v_2 \\
\mathcal{E}(\texttt{raise}\ e) &= \texttt{match}\ \mathcal{E}(e)\ \texttt{with}\ E\,x \to E\,x \mid V\,v \to E\,v \\
\mathcal{E}(\texttt{try}\ e_1\ \texttt{with}\ x \to e_2) & \\
&= \texttt{match}\ \mathcal{E}(e_1)\ \texttt{with}\ E\,x \to \mathcal{E}(e_2) \mid V\,v \to V\,v
\end{aligned}
$$

The transformation propagates error results “upward”, except for `try...with`, which handles the error result.

Alternative: “double-barreled” CPS

Two continuations: `k1` to return a value, `k2` to raise an exception.

```
let safe_add x y k1 k2 =
  let z = x + y in
  if (z lxor x) land (z lxor y) < 0
  then k2 "overflow"
  else k1 z

let rec safe_add_list l k1 k2 =
  match l with
  | [] -> k1 0
  | x :: l ->
      safe_add_list l (fun v -> safe_add x v k1 k2) k2
```

A double-barreled CPS transformation

$$
\begin{aligned}
\mathcal{C}^2(cst) &= \lambda k_1.\, \lambda k_2.\, k_1\ cst \\
\mathcal{C}^2(x) &= \lambda k_1.\, \lambda k_2.\, k_1\ x \\
\mathcal{C}^2(\lambda x.\, e) &= \lambda k_1.\, \lambda k_2.\, k_1\ (\lambda x.\, \mathcal{C}^2(e)) \\
\mathcal{C}^2(e_1\ e_2) &= \lambda k_1.\, \lambda k_2.\, \mathcal{C}^2(e_1)\ (\lambda v_1.\, \mathcal{C}^2(e_2)\ (\lambda v_2.\, v_1\ v_2\ k_1\ k_2)\ k_2)\ k_2 \\
\mathcal{C}^2(\texttt{raise}\ e) &= \lambda k_1.\, \lambda k_2.\, \mathcal{C}^2(e)\ k_2\ k_2 \\
\mathcal{C}^2(\texttt{try}\ e_1\ \texttt{with}\ x \to e_2) & \\
&= \lambda k_1.\, \lambda k_2.\, \mathcal{C}^2(e_1)\ k_1\ (\lambda x.\, \mathcal{C}^2(e_2)\ k_1\ k_2)
\end{aligned}
$$

The transformation propagates the error continuation k_2 “downward” (towards sub-expressions), except for `try...with`, which installs a new error continuation.

Double-barreled CPS transformation $\approx$ ERS transformation followed by CPS transformation

For a program of a base type τ:

$$
\begin{tikzcd}
 & & \tau \arrow[dl, "ERS"'] \arrow[ddr, "CPS^2"] & \\
 & \tau + \mathtt{exn} \arrow[dl, "CPS"'] & & \\
((\tau + \mathtt{exn}) \to \mathit{Res}) \to \mathit{Res} \arrow[rrr, phantom, "\approx"] & & & (\tau \to \mathit{Res}) \to (\mathtt{exn} \to \mathit{Res}) \to \mathit{Res}
\end{tikzcd}
$$

Same type isomorphism as $(A + B) \to C \approx (A \to C) \times (B \to C)$.

Effects and effect handlers

Effects and effect handlers

Algebraic effects: (Plotkin, Power, Pretnar, 2003, 2009)

A theory of the generation, propagation and specification of effects in programming languages.
(Effects = mutable state, I/O, exceptions, non-determinism, …).

($\rightarrow$ Lecture #6)

User-defined effects and effect handlers: (Bauer & Pretnar, 2015)

A powerful control structure inspired by the theory of algebraic effects.

Combines restartable exceptions with delimited continuations.

Catching errors using exceptions

```
type exn += Conversion_failure of string

let int_of_string s =
  match int_of_string_opt s with
  | Some n -> n
  | None  -> raise (Conversion_failure s)

let sum_stringlist lst =
  lst |> List.map int_of_string |> List.fold_left (+) 0

let safe_sum_stringlist lst =
  match sum_stringlist lst with
  | res -> res
  | exception Conversion_failure s ->
     printf "Bad input: %s\n" s; max_int
```

Fixing errors using effects

```ocaml
type _ eff += Conversion_failure : string -> int eff

let int_of_string s =
  match int_of_string_opt s with
  | Some n -> n
  | None  -> perform (Conversion_failure s)

let sum_stringlist lst =
  lst |> List.map int_of_string |> List.fold_left (+) 0

let safe_sum_stringlist lst =
  match sum_stringlist lst with
  | res -> res
  | effect Conversion_failure s, k ->
      printf "Bad input: %s, replaced with 0\n" s;
      continue k 0
```

Example of execution

Without the effect handler: behaves like an uncaught exception.

```
# let n = sum_stringlist ["1"; "xxx"; "2"; "yyy"]
Exception: Stdlib.Effect.Unhandled(Conversion_failure("xxx"))
```

With the effect handler: errors are caught and fixed.

```
# let n = safe_sum_stringlist ["1"; "xxx"; "2"; "yyy"]
Bad input xxx, replaced with 0
Bad input yyy, replaced with 0
val n : int = 3
```

(Examples written and run in OCaml 5.1.1 + an experimental syntax `match with effect`. To use: `opam switch create 5.1.1+effect-syntax` .)

Effects and continuations

```
let int_of_string s = ... perform (Conversion_failure s)

let safe_sum_stringlist lst =
  match ...
  with effect Conversion_failure s, k -> ... continue k 0
```

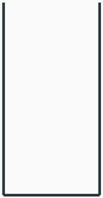
When `perform` raises an effect, its (delimited) continuation is captured and given to the handler along with the effect value.

The effect handler can either discard this continuation `k`,
or restart it on a value of the type expected by the context of the `perform` (here, `int`).

Limitation (in OCaml, not in other languages):
the continuation is "one-shot" (linear) and must be restarted or discarded exactly once.

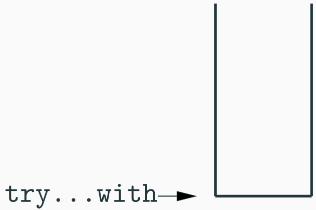
Intuitions in terms of call stacks

Raising an exception = cutting the stack.



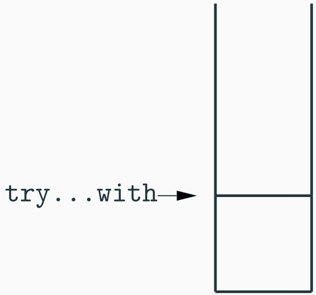
Intuitions in terms of call stacks

Raising an exception = cutting the stack.



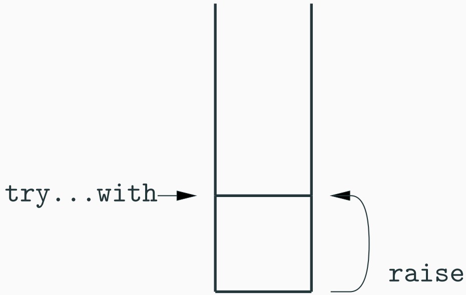
Intuitions in terms of call stacks

Raising an exception = cutting the stack.



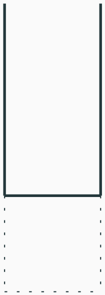
Intuitions in terms of call stacks

Raising an exception = cutting the stack.



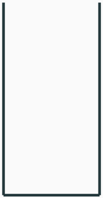
Intuitions in terms of call stacks

Raising an exception = cutting the stack.



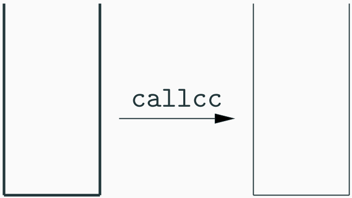
Intuitions in terms of call stacks

Naive undelimited continuations = stack copies (to the heap).



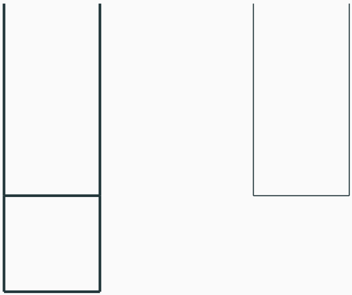
Intuitions in terms of call stacks

Naive undelimited continuations = stack copies (to the heap).



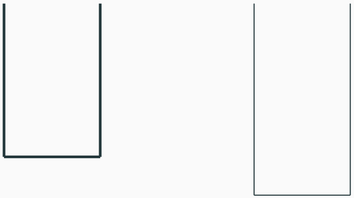
Intuitions in terms of call stacks

Naive undelimited continuations = stack copies (to the heap).



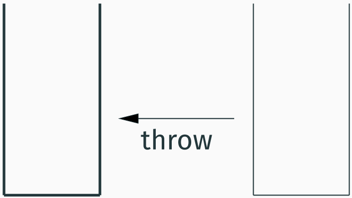
Intuitions in terms of call stacks

Naive undelimited continuations = stack copies (to the heap).



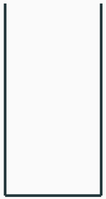
Intuitions in terms of call stacks

Naive undelimited continuations = stack copies (to the heap).



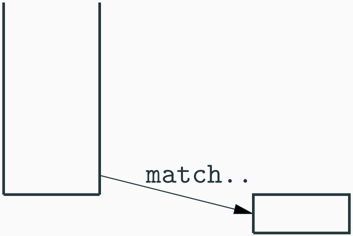
Intuitions in terms of call stacks

Effect handling = switching between several stacks.



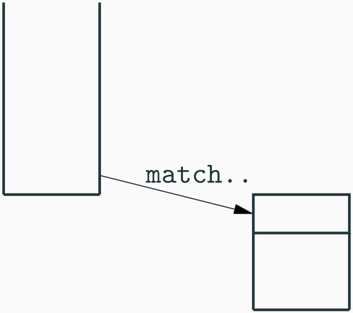
Intuitions in terms of call stacks

Effect handling = switching between several stacks.



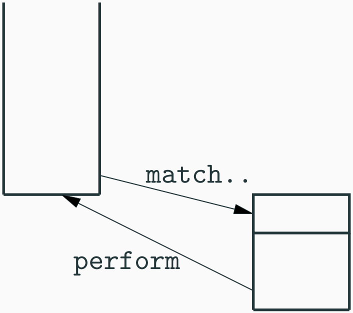
Intuitions in terms of call stacks

Effect handling = switching between several stacks.



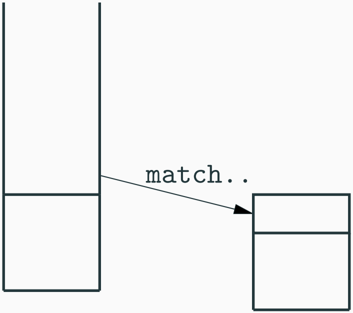
Intuitions in terms of call stacks

Effect handling = switching between several stacks.



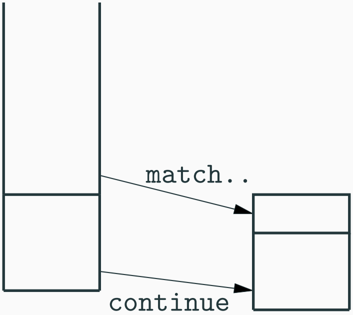
Intuitions in terms of call stacks

Effect handling = switching between several stacks.



Intuitions in terms of call stacks

Effect handling = switching between several stacks.



In OCaml: no stack copying $\rightarrow$ one-shot continuations.

Deep handlers, shallow handlers

Deep handler:

remains in place when a continuation is restarted;
disappears only when the computation terminates normally.

```
# let n = safe_sum_stringlist ["1"; "xxx"; "2"; "yyy"]
Bad input xxx, replaced with 0
Bad input yyy, replaced with 0
val n : int = 3
```

Shallow handler:

disappears as soon as an effect is handled.

```
# let n = safe_sum_stringlist ["1"; "xxx"; "2"; "yyy"]
Bad input xxx, replaced with 0
Exception: Stdlib.Effect.Unhandled(Conversion_failure("yyy"))
```

(In OCaml: `match...with` is “deep”; the `Effect.Shallow` library implements the “shallow” semantics.)

Control inversion on an iterator

As in lecture #4, we assume given an "internal" iterator such as the one over binary trees:

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
let rec tree_iter (f: 'a -> unit) (t: 'a tree) =
  match t with
  | Leaf -> ()
  | Node(l, x, r) -> tree_iter f l; f x; tree_iter f r
```

We'd like to implement an "external" iterator on top of `tree_iter`:

```
type 'a enum = Done | More of 'a * (unit -> 'a enum)
val tree_enum : 'a tree -> 'a enum
```

Control inversion on an iterator

```
let tree_enum (type elt) : elt tree -> elt enum =
  let module Inv = struct
    type _ eff += Next : elt -> unit eff
    let tree_enum (t: elt tree) : elt enum =
      match tree_iter (fun x -> perform (Next x)) t with
      | () -> Done
      | effect Next x, k -> More(x, fun () -> continue k ())
  end in
  Inv.tree_enum
```

We use OCaml's local modules to declare an effect `Next` that is local to the function and has the right type to make `tree_enum` polymorphic in the type `elt` of elements.

Control inversion on an iterator

```
let tree_enum (type elt) : elt tree -> elt enum =
  let module Inv = struct
    type _ eff += Next : elt -> unit eff
    let tree_enum (t: elt tree) : elt enum =
      match tree_iter (fun x -> perform (Next x)) t with
      | () -> Done
      | effect Next x, k -> More(x, fun () -> continue k ())
  end in
  Inv.tree_enum
```

For each element `x` of the tree, the effect `Next x` is performed. The handler receives `x` and the continuation `k` that restarts the traversal.

Control inversion on an iterator

```
let tree_enum (type elt) : elt tree -> elt enum =
  let module Inv = struct
    type _ eff += Next : elt -> unit eff
    let tree_enum (t: elt tree) : elt enum =
      match tree_iter (fun x -> perform (Next x)) t with
      | () -> Done
      | effect Next x, k -> More(x, fun () -> continue k ())
  end in
  Inv.tree_enum
```

When the traversal is over, `tree_iter` returns `()`, which is turned into `Done` by the effect handler.

Control inversion on an iterator

```
let tree_enum (type elt) : elt tree -> elt enum =
  let module Inv = struct
    type _ eff += Next : elt -> unit eff
    let tree_enum (t: elt tree) : elt enum =
      match tree_iter (fun x -> perform (Next x)) t with
      | () -> Done
      | effect Next x, k -> More(x, fun () -> continue k ())
  end in
  Inv.tree_enum
```

Note that the handler changes the type of the computation:

`tree_iter ... t` has type `unit`,

`match tree_iter ...` has type `elt enum`.

Comparing `callcc` with effect handling

Using `callcc`: (lecture #4)

```
callcc (fun k ->
  tree_iter
    (fun x ->
      callcc
        (fun k' ->
           k (More(x, k'))))
    t;
    Done)
```

Two `callcc`: one to exit, one to support restarting.

`More(x, ...)` is computed in the iterated function.

Using effect handling:

```
match
  tree_iter
    (fun x -> perform (Next x))
    t
with
| () -> Done
| effect Next x, k ->
    More(x, fun () -> resume k ())
```

A single `perform` to exit while capturing the restart continuation.

`More(x, ...)` is computed in the handler.

Control inversion on an iterator

This construction can be generalized to invert any internal iterator on any collection type:

```
let enum_of_iter
      (type elt) (type collection)
      (iter: (elt -> unit) -> collection -> unit)
      : collection -> elt enum =
  let module Inv = struct
    type _ eff += Next : elt -> unit eff
    let enum coll =
      match iter (fun x -> perform (Next x)) coll with
      | () -> Done
      | effect Next x, k -> More(x, fun () -> continue k ())
  end in Inv.enum
```

Transforming and re-emitting effects

(M. Pretnar, *An introduction to algebraic effects and handlers*, 2015.)

An effect `Print` for outputting a string.

```
type _ eff += Print : string -> unit eff

let print s = perform (Print s)

let abc () = print "a"; print "b"; print "c"
```

Transforming and re-emitting effects

The effect can be handled as a “true” output on the terminal:

```
let output f =
  match f () with
  | () -> print_newline()
  | effect Print s, k -> print_string s; continue k ()
```

But we can also collect all outputs in a string:

```
let collect f =
  match f () with
  | () -> ""
  | effect Print s, k -> s ^ continue k ()
```

`collect abc` produces the string `"abc"`.

Transforming and re-emitting effects

We can also re-emit the `Print` effect after processing it, for instance to reverse the order of outputs:

```
let reverse f =
  match f () with
  | () -> ()
  | effect Print s, k -> continue k (); print s
```

or to add a sequence number:

```
let number f =
  begin match f () with
  | () -> (fun lineno -> ())
  | effect Print s, k ->
      (fun lineno ->
         print (sprintf "%d:%s\n" lineno s);
         continue k () (lineno + 1))
  end 1
```

Implementing cooperative threads with effects and handlers

A library for cooperative threading

The natural interface in “direct style”:

`spawn: (unit -> unit) -> unit`
: Start a new thread.

`yield: unit -> unit`
: Suspend the current thread;
switch to another runnable thread.

`terminate: unit -> unit`
: Stop the current thread forever.

Defining the corresponding effects

The three operations are defined trivially as raising effects (which will be handled by the scheduler).

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

let spawn f = perform (Spawn f)
let yield () = perform Yield
let terminate () = perform Terminate
```

The state of the scheduler

A queue of threads that were suspended by a call to `yield`, ready to be restarted.

```
let runnable : (unit -> unit) Queue.t = Queue.create()

let suspend f = Queue.add f runnable

let restart () =
  match Queue.take_opt runnable with
  | None -> ()
  | Some f -> f ()
```

The scheduler

```
let rec run (f: unit -> unit) =
  match f() with
  | () -> restart ()
  | effect Terminate, k -> discontinue k; restart ()
  | effect Yield, k -> suspend (continue k); restart ()
  | effect Spawn f, k -> suspend (continue k); run f
```

The scheduler

```
let rec run (f: unit -> unit) =
  match f() with
  | () -> restart ()
  | effect Terminate, k -> discontinue k; restart ()
  | effect Yield, k -> suspend (continue k); restart ()
  | effect Spawn f, k -> suspend (continue k); run f
```

The current thread terminates normally:
we restart another thread.

The scheduler

```
let rec run (f: unit -> unit) =
  match f() with
  | () -> restart ()
  | effect Terminate, k -> discontinue k; restart ()
  | effect Yield, k -> suspend (continue k); restart ()
  | effect Spawn f, k -> suspend (continue k); run f
```

The current thread called `terminate`:
we “discontinue” (throw away) the continuation `k` (the thread will never restart) and we restart another thread.

The scheduler

```
let rec run (f: unit -> unit) =
  match f() with
  | () -> restart ()
  | effect Terminate, k -> discontinue k; restart ()
  | effect Yield, k -> suspend (continue k); restart ()
  | effect Spawn f, k -> suspend (continue k); run f
```

The current thread called `yield`:
we store the continuation `k` as ready to restart,
and we restart another thread.

The scheduler

```
let rec run (f: unit -> unit) =
  match f() with
  | () -> restart ()
  | effect Terminate, k -> discontinue k; restart ()
  | effect Yield, k -> suspend (continue k); restart ()
  | effect Spawn f, k -> suspend (continue k); run f
```

The current thread called `spawn f`:
we store the continuation `k` as ready to restart,
and we start to execute `f`.

The scheduler

```
let rec run (f: unit -> unit) =
  match f() with
  | () -> restart ()
  | effect Terminate, k -> discontinue k; restart ()
  | effect Yield, k -> suspend (continue k); restart ()
  | effect Spawn f, k -> suspend (continue k); run f
```

Alternative:

```
  | effect Spawn f, k ->
      suspend (fun () -> run f); continue k ()
```

In both cases, we must do `run f`, and not just `f()`,
so that the effects of `f()` are handled.

Example of use

A client of the library, written in direct style:

```
let task name n =
  for i = 1 to n do printf "%s%d " name i; yield() done

let _ =
  run (fun () ->
    spawn (fun () -> task "a" 6);
    spawn (fun () -> task "b" 3);
    task "c" 4)
```

Prints `a1 b1 a2 c1 b2 a3 c2 b3 a4 c3 a5 c4 a6`

Adding message-passing communication

`new_channel : unit -> 'a channel`

Create a new channel to pass values of type `'a`.

`recv : 'a channel -> 'a`

Receive a message from the given channel.

`send : 'a channel -> 'a -> unit`

Send the given message on the given channel.

We choose to implement “rendez-vous” semantics (π-calculus):

`send ch v` blocks until another thread calls `recv ch`;

both threads restart;

`recv ch` returns value `v`.

Structure of a communication channel

A channel = two queues,
one for threads blocked on a `send` waiting for a matching `recv`,
the other for threads blocked on a `recv` waiting for a `send`.

```
type 'a channel = {
    senders: ('a * (unit, unit) continuation) Queue.t;
    receivers: ('a, unit) continuation Queue.t
  }

let new_channel () =
  { senders = Queue.create(); receivers = Queue.create() }
```

At any time, at least one of the two queues is empty.

Message-sending operations

As always, whenever we have operations that cannot be implemented locally and must be handled by the scheduler, we turn these operators into effects.

```
type _ eff +=
  | Send : 'a channel * 'a -> unit eff
  | Recv : 'a channel -> 'a eff

let send ch v = perform (Send(ch, v))
let recv ch = perform (Recv ch)
```

The scheduler extended with message passing

```
let rec run (f: unit -> unit) =
  match f () with
    ...
  | effect Send(ch, v), k ->
      begin match Queue.take_opt ch.receivers with
      | Some rc -> suspend (continue k); continue rc v
      | None    -> Queue.add (v, k) ch.senders; restart()
      end
  | effect Recv ch, k ->
      begin match Queue.take_opt ch.senders with
      | Some(v, sn) -> suspend (continue sn); continue k v
      | None        -> Queue.add k ch.receivers; restart()
      end
```

Semantics of effect handlers

A small functional languages with effects and handlers

Expressions:

$$\begin{aligned} e ::= {} & cst \mid x \mid \lambda x.\, e \mid e_1\, e_2 \\ & \mid \texttt{perform}\ e && \text{perform effect } e \\ & \mid \texttt{handle}\ e\ \texttt{with}\ e_{ret}, e_{eff} && \text{handle effects in } e \end{aligned}$$

`perform` e stops evaluation and branches to the nearest enclosing `handle`.

A small functional languages with effects and handlers

Expressions:

$$
\begin{array}{lll}
e ::= cst \mid x \mid \lambda x.\, e \mid e_1\, e_2 & & \\
\quad \mid \texttt{perform}\ e & & \text{perform effect } e \\
\quad \mid \texttt{handle}\ e\ \texttt{with}\ e_{ret}, e_{eff} & & \text{handle effects in } e
\end{array}
$$

$\texttt{handle}\ e\ \texttt{with}\ e_{ret}, e_{eff}$ evaluates the body e.

If e evaluates to value v without performing effects, we apply e_{ret} to v.

If e performs effect f, we apply e_{eff} to (f, k) where f is the value of the effect and k the continuation of the `perform`.

Encoding `match...with effect`

Adding extensible algebraic datatypes and pattern-matching, we can encode

$$
\begin{array}{l}
\texttt{match } e \texttt{ with} \\
\mid x \rightarrow e_0 \\
\mid \texttt{effect } F_1\, x_1, k \rightarrow e_1 \\
\vdots \\
\mid \texttt{effect } F_n\, x_n, k \rightarrow e_n
\end{array}
$$

as

$$
\begin{array}{l}
\texttt{handle } e \texttt{ with} \\
\quad (\lambda x.\, e_0), \\
\quad (\lambda (f, k).\ \texttt{match } f \texttt{ with} \\
\qquad \mid F_1\, x_1 \rightarrow e_1 \mid \ldots \mid F_n\, x_n \rightarrow e_n \\
\qquad \mid \_ \rightarrow k\, (\texttt{perform } f))
\end{array}
$$

Reduction semantics

(Very close to the reduction semantics for exceptions here.)

Two head-reduction rules for`handle`:

$$\begin{aligned}\texttt{handle}\ v\ \texttt{with}\ e_1, e_2 &\xrightarrow{\varepsilon} e_1\, v \\ \texttt{handle}\ D[\texttt{perform}\ v]\ \texttt{with}\ e_1, e_2 &\xrightarrow{\varepsilon} e_2\,(v, (\lambda v'. D[v']))\end{aligned}$$

Here, D is a context with no `handle` enclosing the hole:

Reduction contexts:

$$C ::= [\,] \mid C\ e \mid v\ C \mid \texttt{perform}\ C \mid \texttt{handle}\ C\ \texttt{with}\ e_1, e_2$$

Effect propagation contexts:

$$D ::= [\,] \mid D\ e \mid v\ D \mid \texttt{perform}\ D$$

Deep handlers, shallow handlers

$$\begin{aligned}&\texttt{handle}\ D[\texttt{perform}\ v]\ \texttt{with}\ e_1, e_2\\ &\qquad \xrightarrow{\varepsilon}\ e_2\,(v, \lambda v'.\, D[v'])\end{aligned}$$

The rule above implements shallow handling: the handler is no longer active when the continuation D is restarted.

Deep handling is obtained by reinstalling the handler around the continuation D:

$$\begin{aligned}&\texttt{handle}\ D[\texttt{perform}\ v]\ \texttt{with}\ e_1, e_2\\ &\qquad \xrightarrow{\varepsilon}\ e_2\,(v,\ \lambda v'.\, \texttt{handle}\ D[v']\ \texttt{with}\ e_1, e_2)\end{aligned}$$

CPS transformation for delimited continuations

(M. Materzok, D. Biernacki, *Subtyping delimited continuations*, 2011.)

For undelimited continuations (`callcc`), a CPS-transformed term takes a continuation k as argument, and ensures that

$$\mathcal{C}(e)\ k \xrightarrow{*} k\ cst \quad \text{if} \quad e \xrightarrow{*} cst$$

For delimited continuations, a CPS-transformed term takes $n+1$ continuations $k_0, \ldots, k_n$ as arguments,
where n is the number of enclosing delimiters,
and each k_i is the continuation up to the next delimiter.

$$\mathcal{C}(e)\ k_0\ k_1\ \ldots\ k_n \xrightarrow{*} k_0\ cst\ k_1\ \ldots\ k_n \quad \text{if} \quad e \xrightarrow{*} cst$$

CPS transformation for the pure subset of the language

$$\mathcal{C}(cst) = \lambda k.\ k\ cst$$
$$\mathcal{C}(x) = \lambda k.\ k\ x$$
$$\mathcal{C}(\lambda x.\ e) = \lambda k.\ k\ (\lambda x.\ \mathcal{C}(e))$$
$$\mathcal{C}(e_1\ e_2) = \lambda k.\ \mathcal{C}(e_1)\ (\lambda v_1.\ \mathcal{C}(e_2)\ (\lambda v_2.\ v_1\ v_2\ k))$$

Same definitions as for the usual CBV-value CPS transformation. These definitions remain correct when $\mathcal{C}(e)$ is applied to n continuations, e.g.

$$\mathcal{C}(cst)\ k_0\ k_1\ \ldots\ k_n = (\lambda k.\ k\ cst)\ k_0\ k_1\ \ldots\ k_n \rightarrow k_0\ cst\ k_1\ \ldots\ k_n$$

CPS transformation for delimited continuations

We formalize the operators $\texttt{shift}_0$ and $\texttt{reset}_0$
(O. Danvy and A. Filinksi, 1989).

A delimiter adds a trivial continuation at the head of the list:

$$\mathcal{C}(\texttt{delim}\ e) = \mathcal{C}(e)\ (\lambda x. \lambda k.\ k\ x)$$

so that, in the case where $e \xrightarrow{*} cst$,

$$\begin{array}{rcl} \mathcal{C}(\texttt{delim}\ e)\ k_0\ k_1\ \ldots\ k_n & = & \mathcal{C}(e)\ (\lambda x. \lambda k.\ k\ x)\ k_0\ \ldots\ k_n \\ & \xrightarrow{*} & (\lambda x. \lambda k.\ k\ x)\ cst\ k_0\ \ldots\ k_n \\ & \rightarrow & k_0\ cst\ k_1\ \ldots\ k_n \end{array}$$

CPS transformation for delimited continuations

Symmetrically, the capture operator reifies the first continuation to a value, and removes it from the list:

$$\mathcal{C}(\texttt{capture}\ (\lambda k.e)) = \lambda k.\, \mathcal{C}(e)$$

so that

$$\mathcal{C}(\texttt{capture}\ (\lambda k.\, e))\ k_0\ k_1\ \ldots\ k_n = \mathcal{C}(e)[k \leftarrow k_0]\ k_1\ \ldots\ k_n$$

The evaluation of e continues with k_1, the continuation “after” the nearest delimiter.

The continuation up to this delimiter, k_0, is captured as the k parameter to e.

CPS transformation for effect handlers

(D. Hillerström, S. Lindley, R. Atkey, *Effect handlers via generalised continuations*, 2020.)

The previous approach + the "double-barreled" approach: a CPS-transformed term takes $2n + 2$ continuations as arguments, with n = number of enclosing effect handlers.

$$\mathcal{C}(e)\ k_0\ h_0\ k_1\ h_1\ \ldots\ k_n\ h_n$$

The $k_0, \ldots k_n$ delimited continuations are invoked to return values as results.

The $h_0, \ldots h_n$ delimited continuations are invoked to perform effects.

CPS transformation for effects

For the pure subset of the language: we apply the usual CBV CPS transformation rules.

To perform an effect:

$$\mathcal{C}(\texttt{perform }\ e) = \mathcal{C}(e)\ (\lambda f.\ \lambda k.\ \lambda h.\ h\ (f, \lambda x.\ k\ x\ h))$$

e is evaluated to an effect value f.

We capture the normal continuation k, as well as the effect continuation h, and we invoke h, giving it f as the effect value and $k' = \lambda x.\ k\ x\ h$ as the way to resume after `perform`.

(The application of k to h implements deep handling!)

CPS transformation for effects

An effect handler adds a normal continuation and an effect continuation:

$$\mathcal{C}(\texttt{handle } e \texttt{ with } e_1, e_2) = \mathcal{C}(e)\ (\lambda v.\lambda h.\mathcal{C}(e_1)\ v)\ \mathcal{C}(e_2)$$

In the case where $e \xrightarrow{*} cst$,

$$\begin{aligned} & \mathcal{C}(\texttt{handle } e \texttt{ with } e_1, e_2)\ k_0\ h_0\ \ldots k_n\ h_n \\ = \quad & \mathcal{C}(e)\ (\lambda v.\lambda h.\mathcal{C}(e_1)\ v)\ \mathcal{C}(e_2)\ k_0\ h_0\ \ldots k_n\ h_n \\ \xrightarrow{*} \quad & (\lambda v.\lambda h.\mathcal{C}(e_1)\ v)\ cst\ \mathcal{C}(e_2)\ k_0\ h_0\ \ldots k_n\ h_n \\ \xrightarrow{*} \quad & \mathcal{C}(e_1)\ cst\ k_0\ h_0\ \ldots k_n\ h_n \end{aligned}$$

In the case where e performs effect f with continuation k_f, the continuation $C(e_2)$ is applied to (f, k_f) and to the list $k_0\ h_0\ \ldots$

Summary

Summary

Effect handlers provide:

- A control operator that supports programming in direct style with delimited continuations.
- A presentation of delimited control as restartable exceptions, more intuitive than the control operators viewed earlier.
- A new programming style:
  user code performs effects to invoke the services they need;
  these services are realized by an enclosing handler.

References

References

The OCaml version used for the programming examples:

```
opam update && opam switch create 5.1.1+effect-syntax
```

A general introduction to effect handlers:

- Matija Pretnar: *An Introduction to Algebraic Effects and Handlers*, ENTCS 319, 2015. https://doi.org/10.1016/j.entcs.2015.12.003

CPS transformations for effects:

- Daniel Hillerström, Sam Lindley, Robert Atkey: *Effect Handlers via Generalised Continuations*, J. Funct. Program. 30, 2020. https://doi.org/10.1017/S0956796820000040

The implementation of effects in OCaml version 5:

- KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, Anil Madhavapeddy: *Retrofitting Effect Handlers onto OCaml*, PLDI, 2021. https://arxiv.org/abs/2104.00250