Names and modalities for typing effect handlers

Leo White

Jane Street

Reasons to add side-effects

- Your language is pure
  e.g. Haskell
- Your language doesn't have a particular side-effect
  e.g. concurrency
- Your language doesn't track a particular side-effect
  e.g. untracked exceptions

OCaml 5

- OCaml 5 adds support for algebraic effects
- Intended to provide support for control effects – especially concurrency
- OCaml also has unchecked exceptions

Algebraic effects

```
let choose () : bool =
  perform Or

let fail () : 'a =
  perform Fail

let handle_choice f =
  match f () with
  | x -> [x]
  | effect Or, k ->
      continue k true @ continue k false
  | effect Fail -> []
```

Summary

- Managing effectful computations is about managing open terms
- Effects should have names independent of their types
- Modalities that track locality are useful for managing effects

Effectful computations are open terms

Algebraic data

`<data>`

Algebraic data

```
Branch(<data>,
       Branch(Leaf(<data>)),
              Branch(Leaf(<data>), <data>))
```

Algebraic effects

```
let x ← <computation> in
let y ← <computation> in
return <data>
```

Algebraic effects

```
Or(let x ← <computation> in
   let y ← Or(<computation>, Fail()) in
   return <data>,
   <computation>)
```

Algebraic effects

```
let x ← Or(C₁, C₂) in
C₃
```

Algebraic effects

```
Or(let x ← C₁ in
   C₃,
   let x ← C₂ in
   C₃)
```

Generic operations

```
Or(C₁, C₂)
```

can be expressed as:

```
let b ← Or(return true, return false) in
if b then C₁ else C₂
```

Handling effects

```
Or(let x ← C₁ in
   let y ← Or(C₂, Fail()) in
   return D,
   C₃)
```

Handling effects

```
Or(let x ← C₁ in
   Or(let y ← C₂ in
      return D,
      Fail()),
   C₃)
```

Handling effects

Substitute:

`C`	$\Rightarrow$	`let res ← C in` `return [res]`

`Or(C₁, C₂)`	$\Rightarrow$	`let fst ← C₁ in`
		`let snd ← C₂ in`
		`return (fst @ snd)`

`Fail()`	$\Rightarrow$	`return []`

Handling effects

```
Or(let x ← C₁ in
   Or(let y ← C₂ in
      return D,
      Fail()),
   C₃)
```

$\Rightarrow$

```
let fst ←
  let x ← C₁ in
  let fst ←
    let res ←
      let y ← C₂ in
      return D
    in
    return [res]
  in
  let snd ← return [] in
  return (fst @ snd)
in
let snd ←
  let res ← C₃ in
  return [res]
in
fst @ snd
```

Composing effects

```
Or(Print["hello"](
   Or(C₁, Print["goodbye"](Fail()))),
   C₂)
```

Composing effects

First substitute Print

```
Or(Print["hello"](
   Or(C₁,
      Print["goodbye"](Fail()))),
   C₂)
```

$\Rightarrow$

```
let log ← ref [] in
let res ←
  Or(log := "hello :: !log;
     Or(C₁,
        log := "goodbye" :: !log;
        Fail()),
     C₂)
in
return (res, !log)
```

Composing effects

Then substitute Or/Fail

```
let log ← ref [] in
let res ←
  Or(log := "hello :: !log;
     Or(C1,
        log := "goodbye" :: !log;
        Fail()),
     C2)
in
return (res, !log)
```

⇒

```
let res ←
  let log ← ref [] in
  let res ←
    log := "hello :: !log;
    C1
  in
  return (res, !log)
in
return (Some res)
```

Accidental variable capture

```
let rec find p = function
  | [] -> perform Fail
  | x :: xs -> if p x then x else find p xs

let find_opt p l =
  match find p l with
  | x -> Some x
  | effect Fail -> None

find_opt (fun _ -> perform Fail) l
```

Scope extrusion

```
match (fun () -> perform Read) with
| effect Read, k -> continue k v
| x -> x
```

Effects should have names

Approaches to effects vs. approaches to variables

```
match ... perform Foo ... with
| ... -> ...
| effect Foo -> ...
```

vs.

```
let foo = ... in
... foo ...
```

Approaches to effects vs. approaches to variables

- Monads:
  ```
  2.5 ** (the_thing + 5)
  ```
- Monad transformers:
  ```
  the_thing
  ** (the_other_thing + the_other_other_thing)
  ```
- Naive algebraic effects or MTL:
  ```
  the_float ** (the_int + 5)
  ```
- Naive algebraic effects with *shift*:
  ```
  the_float ** (the_int + the_other_int)
  ```

Names

```
type 'a exn = effect
  | Raise : 'a -> .

let find p = function
  | [] -> perform not_found Raise ()
  | x :: xs -> if p x then x else find p xs
```

Names

From this:

```
[ unit exn;
    int state;
      string state ]
```

To this:

```
[ not_found : unit exn;
    counter : int state;
      log : string state ]
```

Renamings

$$[a_1 : x_1; a_2 : x_2; \ldots; a_n : x_n; r \quad / \quad b_1 : x_i; b_2 : x_j; \ldots; b_m : x_k; r]$$

Renamings

```
let find_opt p l =
  match
    find
      (fun x -> effect [not_found:_; r / r] p x)
      l
  with
  | x -> Some x
  | effect not_found Raise () -> None
```

Abstracting effects

```
module My_effect : sig

  type t : effect

  val do_thing : unit -> int [ my : t ]
  val handle :
    (unit -> 'a [ my : t]) -> 'a

end = struct

  type t = int reader

  ...
end
```

Modalities that track locality

Approach 1: Ignore the problem

Effects in OCaml 5

```
# let () = perform (Set 5);;
```

```
Exception: Stdlib.Effect.Unhandled(Set(5))
```

Unchecked exceptions in many languages

```
# let () = raise Not_found;;
```

```
Exception: Not_found
```

Approach 2: Effect contexts

Arrows or computations annotated with an effect context

```
int -[counter : 'a state; async : async]-> int
```

An empty context corresponds to a closed term

```
int -[]-> int
```

Effect polymorphism

```
val map :
  ('a -['p]-> 'b) -> 'a list -['p]-> 'b list
```

Approach 3: (Weak) Higher-order abstract syntax

Higher-order abstract syntax

```
lam : (tm → tm) → tm
app : tm → tm → tm
```

Global modality for closed terms

- $\Box$`tm`
- $\Box$`(tm → tm)`

Approach 3: (Weak) Higher-order abstract syntax

- An unadorned (`int` $\to$ `int`) arrow corresponds to an open term. It might perform any effects in the current scope.
- An arrow under the global modality ($\Box$(`int` $\to$ `int`)) corresponds to a closed term. It performs no effects.
- The body of a handler takes the generic operation as a parameter
- Values that leave a handler must be closed

Local and global modes in OCaml

Values are either local or global

```
val with_file :
  string -> (file @ local -> 'a) -> 'a
```

Local values cannot escape from their enclosing region

```
with_file "filename" (fun file -> file)
Error: this value escapes its region
```

Values built from local values are also local

```
with_file "filename"
  (fun file ->
     let x = (file, 5) in (fun () -> x))
Error: this value escapes its region
```

Approach 3: (Weak) Higher-order abstract syntax

```
type ('e : effect) handler

let get (h : 'a reader handler) =
  perform h Read

val get : 'a reader handler @ local -> 'a

let handle_reader f v =
  match h -> f h with
  | x -> x
  | effect Read, k -> continue k v

val handle_reader :
    ('a reader handler @ local -> 'b)
    -> 'a -> 'b
```

Avoids effect polymorphism

```
val map :
  ('a -> 'b) @ local -> 'a list -> 'b list
```

Approach 4: Contextual modal types

Contextual modal type theory[1,2]

- `[x : tm; y : tm] tm`
- $\lfloor \texttt{t} \rfloor_{\{x \backslash s;\ y \backslash r\}}$

Move between HOAS and contexts as needed

[1] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. "Contextual modal type theory". (2008).

[2] Brigitte Pientka and Ulrich Schöpp. "Semantical Analysis of Contextual Types.". (2020).

Modal effects for OCaml

Effect contexts on schemes

Traditional approach:

$$\forall\alpha.\tau_1 \xrightarrow{\Sigma} \tau_2$$

Effect contexts part of types.

Alternative approach:

$$\forall\alpha.\tau_1 \rightarrow \tau_2\,[\Sigma]$$

Effect contexts part of *type schemes*

Effect contexts on schemes

Typing judgement has two effect contexts

$$\Gamma \vdash e : \tau \mathrel{?} \Sigma_1 \mathrel{!} \Sigma_2$$

Function abstraction

$$\frac{\Gamma; x : \tau_1 \vdash e : \tau_2 \mathbin{?} \epsilon \mathbin{!} \Sigma}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2 \mathbin{?} \Sigma \mathbin{!} \epsilon}$$

```
⊢ perform var Read
    : 'a ? [] ! [var : 'a reader]

⊢ (fun () -> perform var Read)
      : unit -> 'a ? [var : 'a reader] ! []
```

Function application

$$\frac{\Gamma \vdash f : \tau_1 \to \tau_2 \mathrel{?} \Sigma_1 \mathrel{!} \Sigma_2 \qquad \Gamma \vdash e : \tau_1 \mathrel{?} \epsilon \mathrel{!} \Sigma_3}{\Gamma \vdash f\ e : \tau_2 \mathrel{?} \epsilon \mathrel{!} \Sigma_1 \sqcup \Sigma_2 \sqcup \Sigma_3}$$

```
⊢ (fun () -> perform var Read) ()
      : 'a ? [] ! [var : 'a reader]
```

Function application

$$\frac{\Gamma \vdash f : \tau_1 \to \tau_2 \mathbin{?} \Sigma_1 \mathbin{!} \Sigma_2 \qquad \Gamma \vdash e : \tau_1 \mathbin{?} \epsilon \mathbin{!} \Sigma_3}{\Gamma \vdash f\ e : \tau_2 \mathbin{?} \epsilon \mathbin{!} \Sigma_1 \sqcup \Sigma_2 \sqcup \Sigma_3}$$

```
val run_global : (unit -> 'a) -> 'a

run_global (fun () -> perform var Read)

Error: expected expression with effect []
```

Function application

$$\frac{\Gamma \vdash f : \tau_1 \mathbin{@} \mathsf{local} \rightarrow \tau_2 \mathbin{?} \Sigma_1 \mathbin{!} \Sigma_2 \qquad \Gamma \vdash e : \tau_1 \mathbin{?} \Sigma_3 \mathbin{!} \Sigma_4}{\Gamma \vdash f\ e : \tau_2 \mathbin{?} \epsilon \mathbin{!} \Sigma_1 \sqcup \Sigma_2 \sqcup \Sigma_3 \sqcup \Sigma_4}$$

```
val iter :
    ('a -> unit) @ local -> 'a list -> unit

⊢ List.iter (fun s -> perform log Write(s)) l
     : unit ? [] ! [log : string writer]
```

Function application

$$\frac{\Gamma \vdash f : \sigma[\Sigma_1] \to \tau \,?\, \Sigma_2 \,!\, \Sigma_3 \qquad \Gamma \vdash e : \sigma \,?\, \Sigma_1 \,!\, \Sigma_4}{\Gamma \vdash f\ e : \tau \,?\, \epsilon \,!\, \Sigma_2 \sqcup \Sigma_3 \sqcup \Sigma_4}$$

```
let handle_reader (f : _ [var : 'b reader]) v =
  match f () with
  | x -> x
  | effect var Read, k -> continue k v

val handle_reader :
    (() -> 'a [var :  'b reader])
    -> 'b -> 'a
```

Summary

- Managing effectful computations is about managing open terms
- Effects should have names independent of their types
- Modalities that track locality are useful for managing effects