Effects with Shifted Names in OCaml

Antal Spector-Zabusky

Joint work with Leo White

```
expression, e
  ::= x
   |   λ x . e
   |   fix e
   |   e e
   |   let x = e in e
   |   panic
   |   calm e with e
   |   n#op(e,…,e)
   |   continue e e with H
   |   discontinue e with H
   |   continuation e
   |   ƛ a . e
   |   e n
   |   [r] e
```

```
expression, e
    ::= x
     |  λ x . e
     |  fix e
     |  e e
     |  let x = e in e
     |  panic
     |  calm e with e
     |  n#op(e,…,e)
     |  continue e e with H
     |  discontinue e with H
     |  continuation e
     |  ƛ a . e
     |  e n
     |  [r] e
```

```
expression, e
    ::= x
     |  λ x . e
     |  fix e
     |  e e
     |  let x = e in e
     |  panic
     |  calm e with e
     |  n#op(e,…,e)
     |  continue e e with H
     |  discontinue e with H
     |  continuation e
     |  ƛ a . e
     |  e n
     |  [r] e
```

```
expression, e
   ::= x
     | λ x . e
     | fix e
     | e e
     | let x = e in e
     | panic
     | calm e with e
     | n#op(e,…,e)
     | continue e e with H
     | discontinue e with H
     | continuation e
     | ƛ a . e
     | e n
     | [r] e
```

```
expression, e
    ::= x
      | λ x . e
      | fix e
      | e e
      | let x = e in e
      | panic
      | calm e with e
      | n#op(e,…,e)
      | continue e e with H
      | discontinue e with H
      | continuation e
      | ƛ a . e
      | e n
      | [r] e
```

Effects, handlers, and continuations

```
|   n#op(e,…,e)
|   continue e e with H
|   discontinue e with H
|   continuation e
```

Effects, handlers, and continuations

```
Perform | n#op(e,…,e)
        | continue e e with H
        | discontinue e with H
        | continuation e
```

Effects, handlers, and continuations

```
Perform  | n#op(e,…,e)
Handle { | continue e e with H
         | discontinue e with H
         | continuation e
```

Effects, handlers, and continuations

Perform

Handle

New continuation

```
n#op(e,…,e)
continue e e with H
discontinue e with H
continuation e
```

Effects, handlers, and continuations

```
continue k x_0 with
| return x → e_0
| n_1#op(y_1,…,y_i), k′_1 → e_1
| …
| n_m#op(z_1,…,z_j), k′_m → e_m
```

Effects, handlers, and continuations

```
continue k x0 with
| return x → e0
| n1#op(y1,…,yi), k′1 → e1
| …
| nm#op(z1,…,zj), k′m → em
```

Effects, handlers, and continuations

```
continue k x0 with
| return x → e0
| n1#op(y1,…,yi), k′1 → e1
| …
| nm#op(z1,…,zj), k′m → em
```

Effects, handlers, and continuations

```
continue k x₀ with
 | return x → e₀
 | n₁#op(y₁,…,yᵢ), k′₁ → e₁
 | …
 | nₘ#op(z₁,…,zⱼ), k′ₘ → eₘ
```

Effects, handlers, and continuations

```
continue k x₀ with
| return x → e₀
| n₁#op(y₁,…,yᵢ), k′₁ → e₁
| …
| nₘ#op(z₁,…,zⱼ), k′ₘ → eₘ
```

Names

$n\#op(x_1,\ldots,x_i)$

Names

```
effect 'a state :=
| get : unit -> 'a
| put : 'a -> unit

counter#get()
seed#put(41)
```

Names

```
effect 'a state :=
| get : unit -> 'a
| put : 'a -> unit

counter#get()
seed#put(41)
```

Names

```
| ƛ a . e
| e n
| [r] e
```

Names

Name abstraction | ƛ *a* . *e*

| *e* *n*

| [*r*] *e*

Names

Name abstraction	|	ƛ a . e
Name application	|	e n
	|	[r] e

Names

Name abstraction	|	ƛ *a* . *e*
Name application	|	*e* *n*
Rename	|	[*r*] *e*

State handler

```
let counter_state s0 f =
  let rec go s k x =
    continue k x with
    | return v            -> v, s
    | counter#get(),    k' -> go s  k' s
    | counter#put(s'), k' -> go s' k' ()
  in
  go s0 (continuation f) ()

val counter_state :
  'a ->
  (unit -{counter: 'a state; 'e}-> 'b) -{'e}->
  'b * 'a
```

State handler

```
let counter_state s0 f =
  let rec go s k x =
    continue k x with
    | return v             -> v, s
    | counter#get(),   k' -> go s  k' s
    | counter#put(s'), k' -> go s' k' ()
  in
  go s0 (continuation f) ()

val counter_state :
  'a ->
  (unit -{counter: 'a state; 'e}-> 'b) -{'e}->
  'b * 'a
```

State handler

```
let counter_state s0 f =
 let rec go s k x =
  continue k x with
  | return v            -> v, s
  | counter#get(),   k' -> go s  k' s
  | counter#put(s'), k' -> go s' k' ()
 in
 go s0 (continuation f) ()

val counter_state :
 'a ->
 (unit -{counter: 'a state; 'e}-> 'b) -{'e}->
 'b * 'a
```

State handler

```
let counter_state s0 f =
  let rec go s k x =
    continue k x with
    | return v            -> v, s
    | counter#get(),   k' -> go s  k' s
    | counter#put(s'), k' -> go s' k' ()
  in
  go s0 (continuation f) ()

val counter_state :
  'a ->
  (unit -{counter: 'a state; 'e}-> 'b) -{'e}->
  'b * 'a
```

State handler
with name abstraction

```
let state (name n) s0 f =
  let rec go s k x =
    continue k x with
    | return v               -> v, s
    | (name n)#get(),   k' -> go s  k' s
    | (name n)#put(s'), k' -> go s' k' ()
  in
  go s0 (continuation f) ()

val state :
  (name n) ->
  'a ->
  (unit -{n: 'a state; 'e}-> 'b) -{'e}->
  'b * 'a
```

State handler with name abstraction

```
let state (name n) s0 f =
  let rec go s k x =
    continue k x with
    | return v                 -> v, s
    | (name n)#get(),    k' -> go s  k' s
    | (name n)#put(s'), k' -> go s' k' ()
  in
  go s0 (continuation f) ()

val state :
  (name n) ->
  'a ->
  (unit -{n: 'a state; 'e}-> 'b) -{'e}->
  'b * 'a
```

State handler with name abstraction

```
let state (name n) s0 f =
  let rec go s k x =
    continue k x with
    | return v             -> v, s
    | (name n)#get(),   k' -> go s  k' s
    | (name n)#put(s'), k' -> go s' k' ()
  in
  go s0 (continuation f) ()

val state :
  (name n) ->
  'a ->
  (unit -{n: 'a state; 'e}-> 'b) -{'e}->
  'b * 'a
```

State handler with name abstraction

```
let state (name n) s0 f =
  let rec go s k x =
    continue k x with
    | return v                -> v, s
    | (name n)#get(),   k' -> go s  k' s
    | (name n)#put(s'), k' -> go s' k' ()
  in
  go s0 (continuation f) ()

val state :
  (name n) ->
  'a ->
  (unit -{n: 'a state; 'e}-> 'b) -{'e}->
  'b * 'a
```

State handler with name abstraction

```
let state (name n) s0 f =
  let rec go s k x =
    continue k x with
    | return v             -> v, s
    | (name n)#get(),   k' -> go s  k' s
    | (name n)#put(s'), k' -> go s' k' ()
  in
  go s0 (continuation f) ()

val state :
  (name n) ->
  'a ->
  (unit -{n: 'a state; 'e}-> 'b) -{'e}->
  'b * 'a
```

Renamings

[n_1:b, n_2:a, n_3:_, R <= m_1:a, m_2:b, R]

Renamings

$$[n_1{:}b,\ \boxed{n_2{:}a},\ n_3{:}\_,\ R\ <=\ \boxed{m_1{:}a},\ m_2{:}b,\ R]$$

Renamings

[n_1:b, n_2:a, n_3:_, R <= m_1:a, m_2:b, R]

Renamings

$[\boxed{n_1:b}, n_2:a, n_3:\_, R <= m_1:a, \boxed{m_2:b}, R]$

Renamings

$[n_1{:}b,\ n_2{:}a,\ n_3{:}\_,\ R <= m_1{:}a,\ m_2{:}b,\ R]$

Renamings

$$[n_1{:}b,\ n_2{:}a,\ n_3{:}\_,\ \boxed{R} <= m_1{:}a,\ m_2{:}b,\ \boxed{R}]$$

Renamings

$[n_1{:}b,\ n_2{:}a,\ n_3{:}\_,\ R <= m_1{:}a,\ m_2{:}b,\ R]$

Renamings

$$[n_1:b,\ n_2:a,\ \boxed{n_3:\_},\ R <= m_1:a,\ m_2:b,\ R]$$

Renamings

$$[n_1:b,\ n_2:a,\ n_3:\_,\ R <= m_1:a,\ m_2:b,\ R]$$

Find with exceptions

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

val find :
  ('a -> bool) -> 'a list -> 'a
```

Find with exceptions

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

val find :
  ('a -> bool) -> 'a list -> 'a
```

```
# find (fun xs -> find even xs = 0)
       [];;
```

Find with exceptions

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

val find :
  ('a -> bool) -> 'a list -> 'a




# find (fun xs -> find even xs = 0)
        [];;
⇒ Exception: Not_found.
```

Find with exceptions

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

val find :
  ('a -> bool) -> 'a list -> 'a
```

```
# find (fun xs -> find even xs = 0)
       [[1;3;5]; [1;2;3]; [0;2;4]];;
```

Find with exceptions

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

val find :
  ('a -> bool) -> 'a list -> 'a
```

```
# find (fun xs -> find even xs = 0)
       [[1;3;5]; [1;2;3]; [0;2;4]];;
⇒ Exception: Not_found.
```

Exceptions as an effect

```
effect exn :=
| raise : unit-> .

let optionally (name n) f x =
  continue (continuation f) x with
  | return v              -> Some v
  | (name n)#raise(), _ -> None

val optionally :
  (name n) ->
  ('a -{n: exn; 'e}-> 'b) ->
  'a -{'e}->
  'b option
```

Find with effects

```
let rec find p = function
  | x :: xs ->
      if [not_found:_, R <= R] (p x)
      then x
      else find p xs
  | [] ->
      not_found#raise()

val find :
  ('a -{'e}-> bool) -> 'a list -{not_found : exn; 'e}-> 'a
```

Find with effects

```
let rec find p = function
  | x :: xs ->
      if [not_found:_, R <= R] (p x)
      then x
      else find p xs
  | [] ->
      not_found#raise()

val find :
  ('a -{'e}-> bool) -> 'a list -{not_found : exn; 'e}-> 'a



# optionally (name not_found)
    (find (fun xs -> find even xs = 0))
          [];;
```

Find with effects

```
let rec find p = function
  | x :: xs ->
      if [not_found:_, R <= R] (p x)
      then x
      else find p xs
  | [] ->
      not_found#raise()

val find :
  ('a -{'e}-> bool) -> 'a list -{not_found : exn; 'e}-> 'a



# optionally (name not_found)
    (find (fun xs -> find even xs = 0))
          [];;
⇒ None
```

Find with effects

```
let rec find p = function
  | x :: xs ->
      if [not_found:_, R <= R] (p x)
      then x
      else find p xs
  | [] ->
      not_found#raise()

val find :
  ('a -{'e}-> bool) -> 'a list -{not_found : exn; 'e}-> 'a



# optionally (name not_found)
    (find (fun xs -> find even xs = 0))
          [[1;3;5]; [1;2;3]; [0;2;4]]);;
```

Find with effects

```
let rec find p = function
  | x :: xs ->
      if [not_found:_, R <= R] (p x)
      then x
      else find p xs
  | [] ->
      not_found#raise()

val find :
  ('a -{'e}-> bool) -> 'a list -{not_found : exn; 'e}-> 'a



# optionally (name not_found)
    (find (fun xs -> find even xs = 0))
          [[1;3;5]; [1;2;3]; [0;2;4]]);;
⇒ not_found#raise()
```

Multiple counters with renamings

```
let total l =
  List.iter (fun i -> counter#put(counter#get() + i)) l

val total :
  int list -{counter: int state; 'e}-> unit


let pet_count cat_list dog_list =
  [cats: c; R <= counter: c; R] (total cat_list);
  [dogs: c; R <= counter: c; R] (total dog_list)

val pet_count :
  int list -> int list -{cats: int state; dogs: int state; 'e}->
  unit
```

Thanks!

Our design is available at
https://github.com/antalsz/ocaml-algebraic-effects

We’re available at Dagstuhl :-)

Questions?