Continuations

Hype for Types

September 23, 2024

Exceptions

Find

```
fun fold f z nil = z
  | fold f z (x::xs) = f(x, fold f z xs)
```

Find

```
fun fold f z nil = z
  | fold f z (x::xs) = f(x, fold f z xs)
```

```
fun find (p : 'a -> bool) (l : 'a list) : 'a option =
  fold
    (fn (x,r) => if p x then SOME x else r)
    NONE l
```

Find

```
fun fold f z nil = z
  | fold f z (x::xs) = f(x, fold f z xs)
```

```
fun find (p : 'a -> bool) (l : 'a list) : 'a option =
  fold
    (fn (x,r) => if p x then SOME x else r)
    NONE l
```

```
fun find' (p : 'a -> bool) (l : 'a list) : 'a option =
  let exception Ret of 'a in
    fold
      (fn (x,_) => if p x then raise Ret x else NONE)
      NONE l
    handle Ret x => SOME x
  end
```

Find

```
fun fold f z nil = z
  | fold f z (x::xs) = f(x, fold f z xs)
```

```
fun find (p : 'a -> bool) (l : 'a list) : 'a option =
  fold
    (fn (x,r) => if p x then SOME x else r)
    NONE l
```

```
fun find' (p : 'a -> bool) (l : 'a list) : 'a option =
  let exception Ret of 'a in
    fold
      (fn (x,_) => if p x then raise Ret x else NONE)
      NONE l
    handle Ret x => SOME x
  end
```

Prod

```
fun fold f z nil = z
  | fold f z (x::xs) = f(x, fold f z xs)
```

```
fun prod p l =
  fold
    op*
    1 l
```

Prod

```
fun fold f z nil = z
  | fold f z (x::xs) = f(x, fold f z xs)
```

```
fun prod p l =
  fold
    op*
    1 l
```

```
fun prod p l =
  let exception Ret of int in
    fold
      (fn (0,_) => raise Ret 0 | (x,acc) => x * acc)
      1 l
    handle Ret i => i
  end
```

Continuations

CPS, but at the type level?

```
(* prod : int list -> (int -> 'a) -> 'a *)
fun prod nil      k = k 1
  | prod (0::_ ) k = k 0
  | prod (x::xs) k = prod xs (fn res => k (x * res))
```

CPS, but at the type level?

```
(* prod : int list -> (int -> 'a) -> 'a *)
fun prod nil       k = k 1
  | prod (0::_ ) k = k 0
  | prod (x::xs) k = prod xs (fn res => k (x * res))
```

Goal

Replace type `int -> 'a` with a *jump point* expecting an `int`.

Conveniently, SML <> SML/NJ

```
signature CONT =
  sig
    type 'a cont
    val letcc : ('a cont -> 'a) -> 'a
    val throw : 'a cont -> 'a -> 'b
    val catch : ('a -> void) -> 'a cont
  end

structure K :> CONT =
  struct
    type 'a cont = 'a SMLofNJ.Cont.cont
    val letcc = SMLofNJ.Cont.callcc (* return *)
    val throw = SMLofNJ.Cont.throw
    val catch = fn f => letcc (absurd o f o letcc o
        throw)
  end
```

Some Rules

$$\frac{\Gamma, k : \tau \ \ \texttt{cont} \vdash e : \tau}{\Gamma \vdash \texttt{letcc} \ \ k \ \ \texttt{in} \ \ e : \tau}$$

$$\frac{\Gamma \vdash k : \tau \ \ \texttt{cont} \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \texttt{throw} \ \ k \ e : \tau'}$$

CPS, but at the type level!

```
(* prod : int list -> int cont -> 'a *)
fun prod nil      k = throw k 1
  | prod (0::_ ) k = throw k 0
  | prod (x::xs) k = prod xs (catch (fn res => throw k
        (x * res)))
```

CPS, but at the type level!

```
(* prod : int list -> int cont -> 'a *)
fun prod nil      k = throw k 1
  | prod (0::_ ) k = throw k 0
  | prod (x::xs) k = prod xs (catch (fn res => throw k
      (x * res)))
```

```
- letcc (fn k => prod [1,2,3] k);
val it = 6 : int
```

Example: values with holes

```
(* sum : int list -> (int, int * int cont) either *)
(* sum [2, 1, 5]  ==> INL 8                       *)
(* sum [2, ~2, 5] ==> INR (~2,K)                  *)
```

Example: values with holes

```
(* sum : int list -> (int, int * int cont) either *)
(* sum [2, 1, 5]  ==> INL 8                       *)
(* sum [2, ~2, 5] ==> INR (~2,K)                  *)
```

```
type result = (int, int * int cont) either

fun aux (L : int list) (k : result cont) : int =
  case L of
    nil    => 0
  | x::xs  => letcc (fn here =>
      if x < 0 then throw k (INR (x,here)) else x
    ) + aux xs k

val sum = fn L => letcc (fn k => INL (aux L k))
```

Example: values with holes

```
(* sum : int list -> (int, int * int cont) either *)
(* sum [2, 1, 5]  ==> INL 8                       *)
(* sum [2, ~2, 5] ==> INR (~2,K)                  *)
```

```
fun sumNonneg L =
  case sum L of
     INL res => SOME res
   | INR _   => NONE
```

Example: values with holes

```
(* sum : int list -> (int, int * int cont) either *)
(* sum [2, 1, 5]  ==> INL 8                        *)
(* sum [2, ~2, 5] ==> INR (~2,K)                   *)
```

```
fun sumNonneg L =
  case sum L of
    INL res => SOME res
  | INR _   => NONE
```

```
fun positives L =
  case sum L of
    INL res     => res
  | INR (n, k) => throw k (Int.abs n)
```

Example: values with holes

```
(* sum : int list -> (int, int * int cont) either *)
(* sum [2, 1, 5]  ==> INL 8                        *)
(* sum [2, ~2, 5] ==> INR (~2,K)                   *)
```

```
local
  val readNum = fn () => valOf (Int.fromString (valOf(
     TextIO.inputLine TextIO.stdIn)))
in
  fun fromUser L =
    case sum L of
      INL res => res
    | INR (x, k) => (
        print ("We got: " ^ Int.toString x ^ " (?) ");
        throw k (readNum ())
      )
end
```

Back to Curry-Howard!

Is this Logical?

`'a * 'b`	$A \wedge B$
`'a + 'b`	$A \vee B$
`'a -> 'b`	$A \supset B$
`unit`	$\top$
`void`	$\bot$
`'a cont`	

Is this Logical?

$$\frac{\Gamma, k : \tau\ \texttt{cont} \vdash e : \tau}{\Gamma \vdash \texttt{letcc}\ k\ \texttt{in}\ e : \tau}$$

'a * 'b	$A \wedge B$
'a + 'b	$A \vee B$
'a -> 'b	$A \supset B$
unit	$\top$
void	$\bot$
'a cont	

$$\frac{\Gamma \vdash k : \tau\ \texttt{cont} \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \texttt{throw}\ k\ e : \tau'}$$

Is this Logical?

'a * 'b	$A \wedge B$
'a + 'b	$A \vee B$
'a -> 'b	$A \supset B$
unit	$\top$
void	$\bot$
'a cont	

$$\frac{\Gamma, \tau \ \texttt{cont} \vdash \tau}{\Gamma \vdash \tau}$$

$$\frac{\Gamma \vdash \tau \ \texttt{cont} \qquad \Gamma \vdash \tau}{\Gamma \vdash \tau'}$$

$$\frac{\Gamma, \neg A \vdash A}{\Gamma \vdash A}$$

$$\frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

Is this Logical?

`'a * 'b`	$A \wedge B$
`'a + 'b`	$A \vee B$
`'a -> 'b`	$A \supset B$
`unit`	$\top$
`void`	$\bot$
`'a cont`	$\neg A$

$$\frac{\Gamma, \tau \;\; \texttt{cont} \vdash \tau}{\Gamma \vdash \tau}$$

$$\frac{\Gamma \vdash \tau \;\; \texttt{cont} \qquad \Gamma \vdash \tau}{\Gamma \vdash \tau'}$$

$$\frac{\Gamma, \neg A \vdash A}{\Gamma \vdash A}$$

$$\frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

Programs are proofs...

Now $\neg A \triangleq$ `'a cont` instead of $\neg A \triangleq$ `'a -> void`.
Recall the helper `val catch : ('a -> void) -> 'a cont`

$$\neg(A \land \neg A)$$

$$\neg(A \lor B) \supset \neg A \land \neg B$$

$$(A \supset B) \supset \neg(A \land \neg B)$$

Programs are proofs...

Now $\neg A \triangleq$ `'a cont` instead of $\neg A \triangleq$ `'a -> void`.
Recall the helper `val catch : ('a -> void) -> 'a cont`

$\neg(A \wedge \neg A)$	`catch (fn (a,na) => throw na a)`
$\neg(A \vee B) \supset \neg A \wedge \neg B$	
$(A \supset B) \supset \neg(A \wedge \neg B)$	

Programs are proofs...

Now $\neg A \triangleq$ `'a cont` instead of $\neg A \triangleq$ `'a -> void`.
Recall the helper `val catch : ('a -> void) -> 'a cont`

$\neg(A \wedge \neg A)$	`catch (fn (a,na) => throw na a)`
$\neg(A \vee B) \supset \neg A \wedge \neg B$	`fn k => (catch (fn a => throw k (INL a)), catch (fn b => throw k (INR b)))`
$(A \supset B) \supset \neg(A \wedge \neg B)$	

Programs are proofs...

Now $\neg A \triangleq$ `'a cont` instead of $\neg A \triangleq$ `'a -> void`.

Recall the helper `val catch : ('a -> void) -> 'a cont`

$\neg(A \wedge \neg A)$	`catch (fn (a,na) => throw na a)`
$\neg(A \vee B) \supset \neg A \wedge \neg B$	`fn k => (catch (fn a => throw k (INL a)), catch (fn b => throw k (INR b)))`
$(A \supset B) \supset \neg(A \wedge \neg B)$	`fn f => catch (fn (a,nb) => throw nb (f a))`

Finally a proof of $A \lor \neg A$

Devil: I have an offer for you. Either I give you a ton of gold, or you give me a ton of gold and I will make you the instructor of H4T.

Finally a proof of $A \vee \neg A$

We prove $P \vee \neg P$ by proving $\neg P$. If you believe me, then we are done. If you don't believe me, then you need to give a counter proof, a.k.a a proof of P. Then we $P \vee \neg P$ by proving P.

Finally a proof of $A \vee \neg A$

We prove $P \vee \neg P$ by proving $\neg P$. If you believe me, then we are done. If you don't believe me, then you need to give a counter proof, a.k.a a proof of P. Then we $P \vee \neg P$ by proving P.

Important Idea

Continuations correspond to *classical logic*!

Classical Proofs!?

Now $\neg A \triangleq$ `'a cont` instead of $\neg A \triangleq$ `'a -> void`.
We'll provide the helper `val catch : ('a -> void) -> 'a cont`[1]

$A \vee \neg A$	
$\neg\neg A \supset A$	
$\neg(A \wedge B) \supset \neg A \vee \neg B$	
$\neg(A \wedge \neg B) \supset A \supset B$	

[1] `fun catch f = letcc (absurd o f o letcc o throw)`

Classical Proofs!?

Now $\neg A \triangleq$ `'a cont` instead of $\neg A \triangleq$ `'a -> void`.
We'll provide the helper `val catch : ('a -> void) -> 'a cont`[1]

$A \vee \neg A$	`letcc (fn nana =>` `  INR (catch (fn a => throw nana (INL a))`
$\neg\neg A \supset A$	
$\neg(A \wedge B) \supset \neg A \vee \neg B$	
$\neg(A \wedge \neg B) \supset A \supset B$	

[1] `fun catch f = letcc (absurd o f o letcc o throw)`

Classical Proofs!?

Now $\neg A \triangleq$ `'a cont` instead of $\neg A \triangleq$ `'a -> void`.
We'll provide the helper `val catch : ('a -> void) -> 'a cont`[1]

$A \vee \neg A$	`letcc (fn nana => INR (catch (fn a => throw nana (INL a))`
$\neg\neg A \supset A$	`fn nna => letcc (fn na => throw nna na)`
$\neg(A \wedge B) \supset \neg A \vee \neg B$	
$\neg(A \wedge \neg B) \supset A \supset B$	

[1] `fun catch f = letcc (absurd o f o letcc o throw)`

Classical Proofs!?

Now $\neg A \triangleq$ `'a cont` instead of $\neg A \triangleq$ `'a -> void`.
We'll provide the helper `val catch : ('a -> void) -> 'a cont`[1]

$A \vee \neg A$	`letcc (fn nana =>` `  INR (catch (fn a => throw nana (INL a))`
$\neg\neg A \supset A$	`fn nna =>` `  letcc (fn na => throw nna na)`

$\neg(A \wedge B) \supset \neg A \vee \neg B$	`fn nab => letcc (fn k =>`
	`  INL (catch (fn a => throw k (`
	`  INR (catch (fn b => throw nab (a,b)))))`

$\neg(A \wedge \neg B) \supset A \supset B$	

[1] `fun catch f = letcc (absurd o f o letcc o throw)`

Classical Proofs!?

Now $\neg A \triangleq$ `'a cont` instead of $\neg A \triangleq$ `'a -> void`.
We'll provide the helper `val catch : ('a -> void) -> 'a cont`[1]

$A \vee \neg A$

```
letcc (fn nana =>
  INR (catch (fn a => throw nana (INL a))
```

$\neg\neg A \supset A$

```
 fn nna =>
  letcc (fn na => throw nna na)
```

$\neg(A \wedge B) \supset \neg A \vee \neg B$

```
fn nab => letcc (fn k =>
  INL (catch (fn a => throw k (
  INR (catch (fn b => throw nab (a,b)))))
```

$\neg(A \wedge \neg B) \supset A \supset B$

```
fn k => fn a =>
  letcc (fn nb => throw k (a,nb))
```

[1] `fun catch f = letcc (absurd o f o letcc o throw)`

Demo: True or Not True?

```
val weird = fn () =>
  let
    val p = K.letcc (fn na => INR (K.catch (K.throw na
        o INL))) : (unit,unit K.cont) Either.either
  in
    case p of
      INL () => print "duh, true is true\n"
    | INR k  => (print "uhhh what?\n"; K.throw k ())
  end
```

Conclusion

- Continuations are useful to program with! They let you alter control flow.

Conclusion

- Continuations are useful to program with! They let you alter control flow.
- Classical logic doesn't hold much proof content.