CPS II

15-150 M21

Lecture 0630
30 June 2021

0 CPS Predicates

Simple predicates

Invariant `p : t pred` iff `p : t -> bool` and `p` is total

0630.0 (cpsPred.sml)

```
3 type 'a pred = 'a -> bool
```

0630.1 (cpsPred.sml)

```
6  fun simpleDivBy (n:int):int pred =
7    fn x => (x mod n)=0
8  val divByTwo0 = simpleDivBy 2
9  val divByThree0 = simpleDivBy 3
10 val divByFour0 = simpleDivBy 4
11 val divByFive0 = simpleDivBy 5
12 val divBySix0 = simpleDivBy 6
```

Filter takes in a `pred`

0630.2 (cpsPred.sml)

```
16 fun filter (p : 'a pred) ([] : 'a list) = []
17   | filter p (x::xs) =
18       if p(x)
19       then x::filter p xs
20       else filter p xs
```

Can CPS-ify it

```
filterCPS0 : 'a pred -> 'a list -> ('a list -> 'b)
  -> 'b
```

REQUIRES: true

ENSURES:

```
filterCPS0 p L k  ≅  k(filter p L)
```

0630.3 (cpsPred.sml)

```
24 fun filterCPS0 (p : 'a pred) ([] : 'a list) k
25     = k []
26   | filterCPS0 p (x::xs) k =
27       if p(x)
28       then filterCPS0 p xs (fn l => k(x::l))
29       else filterCPS0 p xs k
```

Crazy idea:

What if the predicate took in a continuation?

```
(* p : int pred  (i.e. p:int->bool, p total) *)
fun p (x:int):bool = ...
```

Accept: `p(x)` $\Longrightarrow$ `true`

Reject: `p(x)` $\Longrightarrow$ `false`

```
(* p :     int
        -> (unit -> 'a)
        -> (unit -> 'a)
        -> 'a                                *)
fun p x sc fc  = ...
```

Accept: `p x sc fc` $\Longrightarrow$ `sc()`

Reject: `p x sc fc` $\Longrightarrow$ `fc()`

Invariant `P : (t1,t2) cpsPred` iff for all `x,sc,fc`,

`P x sc fc` $\Longrightarrow$ `sc()` or `P x sc fc` $\Longrightarrow$ `fc()`

0630.4 (cpsPred.sml)

```
38 type ('a,'b) cpsPred =
39   'a -> (unit -> 'b) -> (unit -> 'b) -> 'b
```

0630.5 (cpsPred.sml)

```
43 fun cpsDivBy (n:int) : (int,'a) cpsPred =
44       fn x => fn sc => fn fc =>
45       if (x mod n)=0 then sc() else fc()
46 fun divByTwo1 x = cpsDivBy 2 x
47 fun divByThree1 x = cpsDivBy 3 x
48 fun divByFour1 x = cpsDivBy 4 x
49 fun divByFive1 x = cpsDivBy 5 x
50 fun divBySix1 x = cpsDivBy 6 x
```

```
filterCPS1 : ('a,'b) cpsPred -> 'a list -> ('a
list -> 'b) -> 'b
```

REQUIRES: true

ENSURES: `filterCPS1 P L k` evaluates to `k(L')`, where `L'` is the sublist of `L` containing all those `x` such that `P` **accepts** `x`.

Live Coding

0630.6 (cpsPred.sml)

```
54 fun filterCPS1 (P : ('a,'b) cpsPred)
55                ([] : 'a list)
56                (k : 'a list -> 'b)
57     = k []
58   | filterCPS1 P (x::xs) k =
59       P x
60       (fn () =>
61          filterCPS1
62            P
63            xs
64            (fn res => k(x::res)))
65       (fn () => filterCPS1 P xs k)
```

Idea: When the "success" case happens, instead of just passing in a `unit` to the success continuation, let's pass in some kind of data, a "witness" to or "evidence" of the success.

```
(* P :      'a
        -> ('e -> 'b)
        -> (unit -> 'b)
        -> 'b                                  *)
fun p x sc fc  = ...
```

Accept (with evidence e): `P x sc fc` $\Longrightarrow$ `sc(e)`

Reject: `P x sc fc` $\Longrightarrow$ `fc()`

Invariant `P : (t1,et,t2) evidPred` iff for all `x,sc,fc`,

`P x sc fc` $\Longrightarrow$ `sc(e)` for some `e:et` or `P x sc fc` $\Longrightarrow$ `fc()`

0630.7 (cpsPred.sml)

```
73 type ('a,'e,'b) evidPred =
74   'a -> ('e -> 'b) -> (unit -> 'b) -> 'b
```

0630.8 (cpsPred.sml)

```
77 fun evidDivBy (n:int): (int,int,'a) evidPred =
78       fn x => fn sc => fn fc =>
79       if (x mod n)=0 then sc(x div n) else fc()
80 fun divByTwo2 x = evidDivBy 2 x
81 fun divByThree2 x = evidDivBy 3 x
82 fun divByFour2 x = evidDivBy 4 x
83 fun divByFive2 x = evidDivBy 5 x
84 fun divBySix2 x = evidDivBy 6 x
```

One more filter

```
filterCPS2 : ('a,'e,'b) evidPred -> 'a list -> (('
a * 'e) list -> 'b) -> 'b
```

REQUIRES: true

ENSURES: `filterCPS2 P L k` evaluates to `k(L')`, where `L'` consists of all pairs `(x,e)` where `x` is an element of `L` and `P x sc fc` $\Longrightarrow$ `sc(e)`.

Live Coding

0630.9 (cpsPred.sml)

```
88 fun filterCPS2
89       (P : ('a,'e,'b) evidPred) [] k
90       = k []
91   | filterCPS2 P (x::xs) k =
92       P x
93       (fn eL => filterCPS2 P xs
94                    (fn res => k((x,eL)::res)))
95       (fn () => filterCPS2 P xs k)
```

Find Sublist

We'll be working with values

```
P : 'a list pred
```

Goal: given `L : t list` and `p : t list pred`, we want to find some sublist `L'` of `L` such that `p` accepts `L'` (`P L'` $\cong$ `true`).

```
findSublist0 : 'a list pred -> 'a list -> ('a list
 -> 'b) -> (unit -> 'b) -> 'b
```

REQUIRES: true

ENSURES: `findSublist0 p L sc fc` evaluates to `sc L'` for some sublist `L'` of `L` such that `p` accepts `L'`. If there is no such `L'`, then `findSublist0 p L sc fc` $\Longrightarrow$ `fc()`.

Live Coding

0630.10 (sublist.sml)

```
3  fun findSublist0 p [] sc fc =
4        if p [] then sc [] else fc ()
5    | findSublist0 p (x::xs) sc fc =
6        findSublist0
7          (fn l => p(x::l))
8          xs
9          (fn l => sc(x::l))
10         (fn () => findSublist0 p xs sc fc)
```

0630.11 (sublist.sml)

```
14 fun findSublist2
15       (P : ('a list,'a list,'b) evidPred)
16       ([] : 'a list)
17       (sc : 'a list -> 'b)
18       (fc : unit -> 'b)
19       : 'b =
20         P [] sc fc
21   | findSublist2 P (x::xs) sc fc =
22       findSublist2
23         (fn l => P (x::l))
24         xs
25         (fn l => sc(x::l))
           (fn () => findSublist2 P xs sc fc)
```

5-minute break

1 CPS Iteration

Many of our list functions can fit into the following description:

Step through the list. For each element, either: (a) throw the element away, (b) combine the element into our ongoing accumulation, (c) stop the process, succeeding with the current element, (d) stop the process with a failure

We're going to abstract this into a very general CPS function.

Module: CPSIteration

`result` type indicates what to do with an element

aux-library/CPSIterate.sml

```
4 datatype result = Accept
5                 | Keep
6                 | Discard
7                 | Break of string
```

A function `check : t -> result` will "govern" the iteration of a `t list`.

```
For : ('a -> result)
    -> 'a list -> ('a -> 'b -> 'b) -> 'b
    -> ('a -> 'c)
    -> (string -> 'c)
    -> ('b -> 'c)
    -> 'c
```

REQUIRES: `check` is total, `combine x` total for all `x`
ENSURES: `(For check L combine base success panic return)` iterates through `L`, applying `check` to each element. It results in either `return(z)` (where `z` is all the elements `x` such that `check x` is `Keep`, `combined` together with `base`), or `success x` for some `x` in `L` such that `check x` is `Accept`, or `panic s` for some `s` such that `check x` $\cong$ `Break s` for some `x` in `L`.

Live Coding

```
53 fun For (check : 'a -> result)
54         (L : 'a list)
55         (combine : 'a -> 'b -> 'b)
56         (base : 'b)
57         (success : 'a -> 'c)
58         (panic : string -> 'c)
59         (return : 'b -> 'c)
60         : 'c
61  =
62   let
63     fun run ([] : 'a list) (k:'b -> 'c) : 'c =
64           k base
65       | run (x::xs) k =
66           (case (check x) of
67               Accept => success x
68           |     Keep => run xs (k o (combine x))
69           |  Discard => run xs k
70           | (Break s)=> panic s)
71   in
72     run L return
73   end
```

Example: Prime divisors

0630.13 (iterate.sml)

```
10 fun div_check m 0 = Break "Divide by zero"
11   | div_check m n =
12       case (m div n, m mod n) of
13             (1,0) => Accept
14         |   (_,0) => Keep
15         |    _    => Discard
16 fun div_success n = SOME [n]
17 fun div_combine x xs = x::xs
```

0630.14 (iterate.sml)

```
20 fun primedivisors m =
21   For (div_check m) primes div_combine
                       > NONE) SOME
```

0630.15 (iterate.sml)

```
26 fun IGNORE x = raise Fail "Ignored"
27 fun KEEP _ = Keep
28 fun DISCARD _ = Discard
29 fun CASE p x = if p x then Keep else Discard
30 fun OPTCASE p x = case p x of
31                       (SOME _) => Keep
32                     | NONE => Discard
33 val valOf : 'a option -> 'a = Option.valOf
```

Summary

Next Time

Thank you!