Higher Order Functions

More abstract abstractions

15-150 M21

Lecture 0621
21 June 2021

0 Lambda Abstraction

```
(fn x => e)
```

"e, but I haven't decided what `x` should be yet"

Given any expression, you can "lambda abstract" any sub-expression by replacing it with a variable, and then taking that variable in as a function argument. This will result in a more-general expression, which may be utilized in more circumstances.

Example: Abstracting 5*7

```
5*7
: int

(fn y => 5*y)
: int -> int

(fn x => (fn y => x*y))
: int -> (int -> int)

(fn f => (fn x => (fn y => f(x,y))))
: ('a * 'b -> 'c) -> ('a -> ('b -> 'c))
```

A new way to multiply?

Consider this expression from the following slide:

```
(fn x=>(fn y=>x*y)) : int -> (int -> int)
```

This expression behaves a lot like `(op * ):int * int -> int`

0621.0 (currying.sml)

```
val mult = fn x => (fn y => x*y)

val 35 = (op * )(5,7)
val 35 = mult 5 7
  (*    mult 5 7 is parsed as (mult 5) 7       *)
```

What's the difference?

```
(op * ) :  int * int -> int
mult :  int -> int -> int
```

Partial Application

The difference between `mult` and `( op * )` is that `mult` may be "partially applied":

0621.0 (currying.sml)

```
val mult = fn x => (fn y => x*y)
```

0621.0 (currying.sml)

```
val quintuple : int -> int = mult 5

(* SYNTAX ERROR:
val quin = (op * )(5, )
*)
```

The function `mult` is called **curried** (named after computer scientist Haskell Curry). Curried functions take their arguments 'one at a time' and may therefore be partially applied.

Uncurried:

```
val foo
    : int * string * bool -> int list option
    = fn (x,s,b) => e
```

Curried:

```
val foo'
    : int -> string -> bool -> int list option
    = fn x => fn s => fn b => e
```

After the lecture, check out this awesome reference by Mia (one of our TAs):
tinyurl.com/150-hofs-note

Some notes about syntax

- Type arrows *right associate*:

```
t1 -> t2 -> t3 -> t4 -> t5
```

  is parsed as

```
t1 -> (t2 -> (t3 -> (t4 -> t5)))
```

- Function application *left associates*:

```
f x1 x2 x3 x4
```

  is parsed as

```
((((f x1) x2) x3) x4)
```

SML has syntactic sugar for declaring curried functions with `fun`:

0621.1 (currying.sml)

```
fun switch (x:int) (y:int) (b:bool) : int
      = if b then y else x

val defaultToZero : int -> bool -> int
      = switch 0
val pickBinary : bool -> int
      = defaultToZero 1
val 0 = pickBinary false
```

This will be helpful for declaring recursive curried functions.

Module: Permute

github.com/smlhelp/aux-library/blob/main/Permute.sml
github.com/smlhelp/aux-library/blob/main/documentation/permute.

Recall: 'a ord

aux-library/Permute.sml

```
(* INVARIANT: cmp : 'a ord must be a comparison
   function *)
  type 'a ord = 'a * 'a -> order
```

Recall: Monomorphic & Polymorphic uncurried msort

```
(* msort : int list -> int list *)
fun msort [] = []
  | msort [x] = [x]
  | msort L = let val (A,B) = split L
              in merge(msort A, msort B)
              end
(* msort : 'a ord * 'a list -> 'a list *)
fun msort (cmp,[]) = []
  | msort (cmp,[x]) = [x]
  | msort (cmp,L) = let val (A,B) = split L in
         merge(cmp,msort(cmp,A), msort(cmp,B))
              end
```

```
merge : 'a ord -> ('a list * 'a list) -> 'a list
```

REQUIRES: L1 and L2 are sorted with respect to cmp
ENSURES: merge cmp (L1,L2) evaluates to a cmp-sorted permutation of L1@L2

aux-library/Permute.sml

```
  fun merge cmp (L1 : 'a list, []) = L1
    | merge cmp ([], L2 : 'a list):'a list = L2
    | merge (cmp : 'a ord) (x::xs, y::ys) =
        (case cmp(x,y) of
            GREATER => y::merge cmp (x::xs,ys)
          | _       => x::merge cmp (xs,y::ys))
```

aux-library/Permute.sml

```
fun msort (cmp : 'a ord) [] = []
  | msort cmp ([x] : 'a list) = [x]
  | msort cmp L =
      let
        val (A,B) = split L
      in
        merge cmp (msort cmp A, msort cmp B)
      end
```

aux-library/Permute.sml

```
fun msort (cmp : 'a ord) [] = []
  | msort cmp ([x] : 'a list) = [x]
```

Works for `string`s!

0621.2 (sorts.sml)

```
val alphabetize
    : string list -> string list
    = Permute.msort String.compare
```

0621.3 (sorts.sml)

```
fun cmpPair((a,b),(c,d)) =
  case Int.compare(a,c) of
       EQUAL => Int.compare(b,d)
     | z => z

val pairSort
     : (int*int) list -> (int*int) list
      = Permute.msort cmpPair
```

0621.4 (sorts.sml)

```
fun cmpLen (L1 :'a list,L2:'a list) =
    Int.compare(len L1,len L2)

(* val lenSort = msort cmpLen *)
val lenSort : 'a list list -> 'a list list
    = fn L => Permute.msort cmpLen L
```

Note: it's possible for `cmpLen(L1,L2)` $\cong$ `EQUAL` even though `L1 <> L2`. This is where it becomes relevant that mergesort is *stable*!

Write very-general curried functions, and supply them with some of their arguments in order to achieve what we want

```
fun superUseful a b c d e f g = ...

val task1 = superUseful 3 false (fn x=>4+x) []
val task2 = superUseful 5 true
val task3 = superUseful 0 true fact [] "" 7
```

Why?

- Can explicitly codify common patterns
- Fewer functions to write
- Fewer functions to prove correct
- Fewer functions to analyze
- Can swap out the implementation of the general function with a provably-equivalent but more efficient implementation. Then everything is updated to the new version.

5-minute break

1 Higher-Order Functions

Higher Order Function:

Any function which takes a function as an argument or returns a function. (has multiple ->'s in its type)

The idea

In the past section, we codified a particular kind of behavior (sorting a list with respect to some comparison function), and defined a higher-order function which abstractly codifies that process. We'll now do the same with the following kinds of behaviors.

- Applying one function to an argument, and then applying a function to the result
- Applying a function to every element of a list
- Iterating through a list and accumulating a result
- Removing certain elements from a list

Module: Fn

(SMLNJ basis)

(op o) : ('b -> 'c)*('a -> 'b) -> 'a -> 'c
REQUIRES: true
ENSURES: (g o f) $\cong$ h such that h(x) $\cong$ g(f(x)) for all suitably-typed x

0621.5 (hofs.sml)

```
infix o
fun (g o f) x = g(f(x))
(* OR: fun (g o f) = fn x => g(f(x)) *)

val addThree = (fn x => x+3)
val addSix = addThree o addTree
```

Module: List

(SMLNJ basis)

Apply a function to every element of a list

`map : ('a -> 'b) -> 'a list -> 'b list`

REQUIRES: `f` is total

ENSURES: `map f L` evaluates to the list `L'` consisting of `f` applied to each of the elements of `L`

0621.6 (hofs.sml)

```
fun map f [] = []
  | map f (x::xs) = (f x)::map f xs

val [2,3,4] = map (fn x=>x+1) [1,2,3]
val curries =
  map (fn name => name ^ " Curry")
  ["Haskell","Steph","Tim","Denzel"]
```

Let `val f = (fn x => if x mod 2 = 0 then SOME(x div 2) else NONE)`

```
map f [4,5,6,7]
⟹ (f 4):: map f [5,6,7]
⟹ (SOME 2)::map f [5,6,7]
⟹ (SOME 2)::(f 5)::map f [6,7]
⟹ (SOME 2)::(NONE)::map f [6,7]
⟹ (SOME 2)::(NONE)::(f 6)::map f [7]
⟹ (SOME 2)::(NONE)::(SOME 3)::map f [7]
⟹ (SOME 2)::(NONE)::(SOME 3)::(f 7) :: map f []
⟹ (SOME 2)::(NONE)::(SOME 3)::(NONE):: map f []
⟹ (SOME 2)::(NONE)::(SOME 3)::(NONE)::[]
= [SOME 2,NONE,SOME 3,NONE]
```

Step through a list and combine

`foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b`

REQUIRES: g is total

ENSURES: `foldl g acc [x_1,...,x_n]` $\cong$ `g(x_n,g(...,g(x1,acc)...))`

0621.7 (hofs.sml)

```
fun foldl g acc [] = acc
  | foldl g acc (x::xs) = foldl g (g(x,acc)) xs

val sum = foldl (op +) 0
val prod = foldl (op * ) 1
val rev = fn L => foldl (op::) [] L
```

```
foldl (op^) "!" ["H","E","L","L","O"]
⟹ foldl (op^) "H!" ["E","L","L","O"]
⟹ foldl (op^) "EH!" ["L","L","O"]
⟹ foldl (op^) "LEH!" ["L","O"]
⟹ foldl (op^) "LLEH!" ["O"]
⟹ foldl (op^) "OLLEH!" []
⟹ "OLLEH!"
```

Step through a list the other way and combine

```
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

REQUIRES: g is total

ENSURES: `foldr g acc [x_1,...,x_n]` ≅ `g(x_1,g(...,g(xn,acc)...))`

0621.8 (hofs.sml)

```
fun foldr g acc [] = acc
  | foldr g acc (x::xs) = g(x,foldr g acc xs)

val sum = foldr (op +) 0
val prod = foldr (op * ) 1
val concat = foldr (op ^) ""
```

```
foldr (op^) "!" ["H","E","L","L","O"]
⟹ "H"^(foldr (op^) "!" ["E","L","L","O"])
⟹ "H"^("E"^(foldr (op^) "!" ["L","L","O"]))
⟹ "H"^("E"^("L"^(foldr (op^) "!" ["L","O"])))
⟹ "H"^("E"^("L"^("L"^(foldr (op^) "!" ["O"]))))
⟹ "H"^("E"^("L"^("L"^("O"^(foldr (op^) "!" [])))))
⟹ "H"^("E"^("L"^("L"^("O"^"!"))))
⟹ "HELLO!"
```

- Currying allows us to write functions which can be partially applied
- We can abstract common patterns of reasoning into curried HOFs, which can then be partially applied to get functions with more specific behavior
  - Polymorphic sorting
  - Composition
  - Mapping
  - Folding
  - Filtering

Next Time

- Totality and Extensional Equivalence of HOFs
- Staging
- Combinators

Thank you!