Polymorphism

An introduction to abstraction

15-150 M21

Lecture 0616
16 June 2021

- ✓ Basics of Functional Computation
- ✓ Induction and Recursion
- Polymorphism & Higher-Order Functions
- Functional Control Flow
- The SML Modules System
- Applications & Connections

0 Polymorphism

Recall: datatypes

- ✓ The `datatype` keyword is used to declare new datatypes by their constructors
- ✓ Each recursive datatype comes equipped with a method of recursion & principle of structural induction
- We can parametrize datatypes by type variables (e.g. `list`, `option`)

```
datatype 'a list = [] | :: of 'a * 'a list
```

- 'a is a **polymorphic type variable**
- 'a can be **instantiated** to any type, e.g. `int` or `bool list`

```
datatype 'a option = NONE | SOME of 'a
```

Can make our definition of tree polymorphic:

0616.0 (poly.sml)

```
datatype 'a tree =
    Empty | Node of 'a tree * 'a * 'a tree
```

(Note: the version of `tree` on the current homework is *not* polymorphic)

0616.1 (poly.sml)

```
datatype 'a shrub =
    Leaf of 'a | Branch of 'a shrub * 'a shrub
```

Shrubs are like trees, except they store their data at the leaves instead of the nodes

- `type` can also be parametrized:

0616.2 (poly.sml)

```
type 'a ord = 'a * 'a -> order
```

- We can parametrize by multiple type variables:

0616.3 (poly.sml)

```
datatype ('a,'b) either =
    inL of 'a | inR of 'b
```

Question:

What's the type of ::?

Abstraction and Instantiation

Given a **polymorphic type** (a type containing polymorphic type variables), an *instance* of that type is a type arrived at by replacing some of the type variables with more specific types

- `int list` is an instance of `'a list`
- `(int * string) option` is an instance of `('a * string) option` and of `(int * 'b) option`, which are both instances of `('a * 'b) option`
- Every function type is an instance of `'a -> 'b`
- Every type is an instance of `'a`

We write

`(op::) : 'a * 'a list -> 'a list`

to indicate that, **for all types** `t`, `(op::)` is a well-typed expression of type `t * t list -> t list`.

`'a * 'a list -> 'a list` is called the **most general type** of `op::`. We can use `op::` as a value of type `bool*bool list -> bool list` or `string*string list -> string list` because these are *instances of its most general type*.

Key Point:

A well-typed expression can be used at any instance of its most general type

What is the most general type of

```
fun len [] = 0
  | len (_::xs) = 1 + len xs
```

- The MGT of `len` is `'a list -> int`
- This type must be an instance of `len`'s MGT, since `len` can be used as an expression of type `t list -> int` for any type `t` – any instance of `'a list -> int`.
- But we cannot make this type more general: we cannot always use `len` as an expression of type `t -> int` (e.g. if `t=string`), so its MGT cannot be `'a -> int`. It also always returns an integer, so it cannot have MGT `'a list -> 'b`, etc.

Some standard polymorphic functions:

0616.4 (poly.sml)

```
val id : 'a -> 'a = fn x => x

fun fst(x : 'a, y : 'b):'a = x
fun snd(x : 'a, y : 'b):'b = y

fun swap(x : 'a, y : 'b):'b * 'a = (y,x)

val const3 : 'a -> int = fn _ => 3
```

Negative results

- There is no value whose MGT is `'a`
- There is no total function whose MGT is `'a -> 'b`
- There is no total function whose MGT is `int -> 'a`, `string -> 'a`, etc.

Lots of stuff is polymorphic

- `[] : 'a list`
- `(op ::) : 'a * 'a list -> 'a list`
- `len : 'a list -> int`
- `rev : 'a list -> 'a list`
- `trev : 'a list * 'a list -> 'a list`
- `(op @) : 'a list * 'a list -> 'a list`

Key Skill:

Determining the MGT of a polymorphic expression

Example: Finding the MGT

0616.5 (poly.sml)

```
fun ex1 (v,w,x,y,z) =
  if (v w) then x else (y,z)
```

- `ex1` is a function, so it's an instance of `'a -> 'b`
- The input is a 5-tuple

```
('a * 'b * 'c * 'd * 'e) -> 'f
```

- `v` must be a function which can be applied to `w` and returns a bool

```
(('b -> bool) * 'b * 'c * 'd * 'e) -> 'f
```

- `x` must be the same type as `(y,z)`

```
(('b -> bool) * 'b * ('d * 'e) * 'd * 'e) -> 'f
```

0616.5 (poly.sml)

```
fun ex1 (v,w,x,y,z) =
  if (v w) then x else (y,z)
```

- x must be the same type as (y,z)

```
(('b -> bool) * 'b * ('d * 'e) * 'd * 'e) -> 'f
```

- This is also the return type of the function.

```
(('b -> bool) * 'b * ('d * 'e) * 'd * 'e) -> ('d * 'e)
```

- We can stop at this point: check that ex1 can be used at any instance of

Check Your Understanding

Try figuring out the MGT of these expressions. You can then test yourself by entering the expression (or declaration) into the REPL

0616.6 (poly.sml)

```
fun ex2 (x,y) = x(y)
val ex3 = (SOME,NONE)
fun ex4 (a,b,c) = b=[a] andalso c a
fun ex5 (w,x,y) = (w x) y
fun ex6 (a,b,c,d) = a(b,c(d,[]))
```

5-minute break

What's the difference between these statements?

"For all values `L : 'a list`, P(`L`) holds"

"For all types `t` and all values `L : t list`, P(`L`) holds"

You almost always mean the second one. Remember: there's only **one** value of type `'a list`, `[]`.

1 Polymorphic Sorting

Check Your Understanding How many total values are there of type `'a * 'a -> 'a * 'a`?

```
fn (x,y) => (x,y)
fn (x,y) => (y,x)
fn (x,y) => (x,x)
fn (x,y) => (y,y)
```

Thm. These are the only values whose most general type is (at least as general as) `'a * 'a -> 'a * 'a`

Claim: the only values of type `'a list -> 'a list` are functions which *ignore the content of the list* and merely operate on its *structure*

Module: Permute

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

aux-library/Permute.sml

```
(* INVARIANT: f : 'a perm must be total and
 *            bijective on the elements of the list
 *)
  type 'a perm = 'a list -> 'a list
```

aux-library/Permute.sml

```
val rev : 'a perm
```

```
local
  fun trev ([],acc) = acc
    | trev (x::xs,acc) = trev(xs,x::acc)
in
  val rev = fn L => trev(L,[])
end
```

aux-library/Permute.sml

```
  val riffle : 'a perm
```

```
local
  fun interleave (L1,[]) = L1
    | interleave ([],L2) = L2
    | interleave (x::xs,y::ys) =
            x::y::interleave(xs,ys)
  fun cleanSplit L =
    let val n = (len L) div 2
    in (List.take(L,n),List.drop(L,n))
    end
in
  val riffle = fn L => interleave(cleanSplit L)
```

0616.7 (sorting.sml)

```
fun split [] = ([],[])
  | split [x] = ([x],[])
  | split (x::x'::xs) =
      let
          val (A,B) = split xs
      in
        (x::A,x'::B)
      end
```

0616.8 (sorting.sml)

```
fun merge (L1,[]) = L1
  | merge ([],L2) = L2
  | merge (x::xs,y::ys) =
      (case Int.compare(x,y) of
         GREATER => y::merge(x::xs,ys)
       | _       => x::merge(xs,y::ys))
```

Key Idea:
Lambda Abstraction

0616.9 (sorting.sml)

```
fun merge (_, L1 : 'a list, []) = L1
  | merge (_, [], 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))
```

Different Approach:
“Curried” version

aux-library/Permute.sml

```
  val msort : 'a ord -> 'a perm
```

```
  fun msort cmp =
     fn []  => []
      | [x] => [x]
      | L   =>
          let val (A,B) = split L
          in merge(cmp, msort cmp A, msort cmp B)
          end
```

- Polymorphism allows us to abstract functions to operate on various types
- Every well-typed expression has a *most general type*, and can be used at any instance of its MGT
- We can make our sorting functions polymorphic

Next Time

- Lambda abstraction and currying
- Higher-Order Functions

Thank you!