This topic first introduces parametric polymorphism, which allows general type schemes to be given for values that have many different types. It then focuses on higher-order functions in detail, including closures.
Parametric Polymorphism

• In F# (and similar languages) some definitions can be assigned more than one type.

• E.g., the identity function

  ```
  let identity x = x
  ```

can be assigned all of the following types:

  ```
  int -> int    bool list -> bool list    int*int -> int*int
  ```

• This happens because the definition does not depend on some parts of the type
  ◦ identity does nothing with x except return it, so it can be any type.

• Such definitions can be assigned a type scheme which includes type variables (written as 'a, 'b, 'c, ...) which can be replaced by any type.
  ◦ E.g., identity : 'a -> 'a

• Type inference automatically determines the most general type.
  ◦ This is done by assigning each expression a type variable, and then solving constraints between these variables via unification (later).

• Each time a definition with a polymorphic type is used, a type is assigned to that use by replacing each type variables with a type.
Polymorphism examples

Here’s the type scheme for map and two uses of it using different types.

```plaintext
let rec map f =
  function  // (This is usually at the end of the line above, but clearer here.)
  | [] -> []
  | x::xs -> (f x) :: (map f xs)

val map : ('a -> 'b) -> ('a list -> 'b list)  // Only first parens are needed

let mapped1 = map ((+) 1) [1, 2, 3]  // Both 'a and 'b replaced by int
val mapped1 : int list

let mapper2 = map ((=) 1.0)  // 'a replaced by float, 'b replaced by bool
val mapper2 : float list -> bool list
```

Similarly:

```plaintext
rev : 'a list -> 'a list  (@) : 'a list -> 'a list -> 'a list
[] : 'a list
(++) : 'a -> 'a list -> 'a list
```
Higher-order functions

- **First-order functions** only take and return non-functions.
  - E.g., types like \(\text{int} \rightarrow \text{int}\) and \(\text{int} \ast \text{int} \rightarrow \text{int list}\)
  - Many languages only have first-order functions.

- A **second-order function** can take and return first-order fns.
  - E.g., types like \((\text{int} \rightarrow \text{int}) \rightarrow \text{int}\) and \(\text{int} \rightarrow \text{int} \ast (\text{int} \rightarrow \text{int})\)

- **Higher-order functions** can take and return fns. of any order.
  - Functions can also be stored in data structures, etc.

- Higher-order functions are a very powerful feature.
  - They can “glue” functions together to form more complex ones.

- With higher-order functions, functions become just like other types.
  - Functions don’t even need to have names (as for pairs and lists)
  - E.g., \(\text{List.map (fun x -> x \ast x)} [1; 4; 6] \) yields \([1; 16; 36]\)
Function definitions

- Recall that every function definition is equivalent to a one using `fun`:
  ```
  let f x = Expression  is an abbreviation for:
  let f = fun x -> Expression
  ```
  - I.e., function definitions define constants that happen to be functions.

- This extends to functions with many arguments:
  ```
  let f x y = Expression  is the same as
  let f = fun x -> (fun y -> Expression )
  // The parens are only for clarity here – usually they are left out.
  ```
  - Note that `f` is higher-order: it returns a function.
  - When `f` is applied to one argument, the result is a function that can be
    applied to another argument.
  - This is called a *Curried function*, after the logician Haskell Curry.
  - We can pass the result of applying to one argument to a another function –
    e.g suppose we want to add “Dr ” in front of each string in a list:
    ```
    map ((+) "Dr ") ["Who"; "X"]  yields  ["Dr Who"; "Dr X"]
    ```
    - [Here `(+)` makes `+` into a prefix fn with type `string -> string -> string`.]
Closures

- Functions can also be constructed at any point in an expression.

```latex
let mapping1D ((aMin, aMax), (bMin, bMax)): float -> float =
    let aRange = aMax - aMin
    let bRange = bMax - bMin
    fun a -> bMin + bRange*(a-aMin)/aRange
```

- Here the last line returns a function that can be used many times:

```latex
let remap = mapping1D ((0.0, 1.0), (3.0, 8.0))
remap 0.5 /*yields 5.5*/
remap 0.8 /*yields 7.0*/
```

- Note: remap will use the local vars: aMin, aRange, bMin, bRange
  - But mapping1D has already returned!
  - The values of these variables are stored as part of the function that is returned by mapping1D - but normally the programmer can ignore this.
  - This is called a closure, and is a defining feature of functional languages.
  - Many languages now have closures: C#, Python, Perl, JavaScript, VB
  - Closures can be simulated by creating classes, but this is very awkward.
One use of higher order functions is to *abstract* out components

- E.g. by turning a constant into a function parameter so that many different values can be supplied.
- This allows reuse, more concise programs and reduces maintenance.
- Write one function and use it ten times, not ten slightly different ones.

Our previous `qsort` function always used the built-in `$\leq$` and `$>$`.

- Suppose we sometimes need to sort other ways (e.g. strings by length.)
- We could write a different sorting function each time. Or:

  ```haskell
  let rec qsort leq = |
    | [] -> []
    | x1::xs -> qsort leq [for x in xs do if leq x x1 then yield x] @ [x1] @ qsort leq [for x in xs do if not (leq x x1) then yield x]
  |
  qsort ($\leq$) [65;98;2343;343]  // yields [65; 98; 343; 2343]
  qsort ($\Rightarrow$) [65;98;2343;343]  // yields [2343; 343; 98; 65]
  qsort (fun xs-> fun ys-> List.length xs < List.length ys) [[1;2;7];[];[3;4]]
  // yields [[]; [3; 4]; [1; 2; 7]]
  ```
Comparison with objects

- Object-orientation also provides some of the benefits of higher order functions.
  - E.g., an interface for comparison objects can be defined, with many implementing classes.

- Objects can be thought of as just tuples of functions – one for each method.
  - The provide some of the same kinds of hiding and abstraction.
  - Objects are sometimes referred to as “poor-man’s closures” by functional programmers.

- Instance variables correspond roughly to the variables stored in a closure.

[Of course, there is a more imperative flavour to objects also compared to FP.]

- Programming with functions directly allows some kinds of programming that cannot easily be done with objects.
  - E.g., creation of closures easily, higher order function “glue”.