

Labelled & Optional Parameters

Kaung Htet

Labeled Parameters

Labeled Arguments

- In the core language, most functions are specified with positional arguments.
- Labeled arguments are a convenient extension to the core language.
- Can be passed in different order than one of their definitions.
 - Increases flexibility.

~label:pattern

```
# let rec range ~first:a ~last:b =  
  if a > b then []  
  else a :: range ~first:(succ a) ~last:b;;  
  
val range : first:int -> last:int -> int list = <fun>  
  
# range 3 6;;  
- : int = [3;4;5;6]  
  
# range ~first:3 ~second:6;;  
- : int = [3;4;5;6]  
  
# range ~second:6 ~first:3;;  
- : int = [3;4;5;6]
```

Label punning



~f:f

```
# let find l ~f =  
  let rec loop = function  
    | [] -> None  
    | hd :: tl -> if f hd then Some hd else loop tl  
  in  
  loop l;;  
val find : 'a list -> f:('a -> bool) -> 'a option = <fun>  
  
# find ~f:(fun x -> x = 3) [1;2;3];;  
- : int option = Some 3
```

```
# let ratio ~num ~denom = float num /. float denom;;  
val ratio : num:int -> denom:int -> float = <fun>
```

```
# let num = 3 in  
  let denom = 4 in  
  ratio ~num ~denom;;  
- : float = 0.75
```

- When defining a function
 - with lots of arguments
 - with multiple arguments of the same type that might get confused with each other

```
val substring: string -> int -> int -> string
```

- with flexibility on the order which arguments are passed.

```
val substring: string -> pos:int -> len: int ->  
string
```

Inference of labeled args

```
# let foobar ~x ~y ~f =  
  let dx = (f ~x ~y) in  
  let dy = (f ~x ~y) in  
  (dx, dy)
```

```
::
```

```
val foobar : x:'a -> y:'b ->
```

```
f:(x:'a -> y:'b -> 'c) -> 'c * 'c = <fun>
```

$f:(y:'a \rightarrow x:'b \rightarrow 'c)$

Inference of labeled args

```
# let foobar ~x ~y ~f =  
  let dx = (f ~x ~y) in  
  let dy = (f ~y ~x) in  
  (dx, dy)  
;;
```

Error: This function is applied to arguments
in an order different from other calls.
This is only allowed when the real type is
known.

Inference of labeled args

```
# let foobar ~x ~y ~(f: x:'a -> y:'b -> 'c) =  
  let dx = (f ~x ~y) in  
  let dy = (f ~y ~x) in  
  (dx, dy)  
  
;;  
val foobar : x:'a -> y:'b ->  
f:(x:'a -> y:'b -> 'c) -> 'c * 'c = <fun>
```

Provide explicit type information

- By default, standard library functions are not labeled.
- The module `StdLabels` redefines some modules of the standard library with labeled versions of some functions.

StdLabels

StdLabels.List

StdLabels.Array

StdLabels.Bytes

StdLabels.String

t

create

exists

compare

filter

fold_right

map

partition

append

assoc

assq

blit

capitalize

2

3

2

2

2

4

2

3

2

Values

append

assoc

assq

combine

concat

exists

exists2

fast_sort

filter

find

find_all

flatten

fold_left

fold_left2

fold_right

fold_right2

for_all

StdLabels.List

Purchase Dash

List scanning

```

val for_all : f:( 'a -> bool) -> 'a list -> bool
  for_all p [a1; ...; an] checks if all elements of the list satisfy the predicate p. That is, it returns (p a1) && (p a2) && ...
  && (p an).

val exists : f:( 'a -> bool) -> 'a list -> bool
  exists p [a1; ...; an] checks if at least one element of the list satisfies the predicate p. That is, it returns (p a1) || (p a2)
  || ... || (p an).

val for_all2 : f:( 'a -> 'b -> bool) -> 'a list -> 'b list -> bool
  Same as ListLabels.for_all, but for a two-argument predicate. Raise Invalid_argument if the two lists have different lengths.

val exists2 : f:( 'a -> 'b -> bool) -> 'a list -> 'b list -> bool
  Same as ListLabels.exists, but for a two-argument predicate. Raise Invalid_argument if the two lists have different lengths.

val mem : 'a -> set:'a list -> bool
  mem a l is true if and only if a is equal to an element of l.

val memq : 'a -> set:'a list -> bool
  Same as ListLabels.mem, but uses physical equality instead of structural equality to compare list elements.

```

List searching

```

val find : f:( 'a -> bool) -> 'a list -> 'a
  find p l returns the first element of the list l that satisfies the predicate p. Raise Not_found if there is no value that satisfies p in
  the list l.

val filter : f:( 'a -> bool) -> 'a list -> 'a list
  filter p l returns all the elements of the list l that satisfy the predicate p. The order of the elements in the input list is preserved.

val find_all : f:( 'a -> bool) -> 'a list -> 'a list
  find_all is another name for ListLabels.filter.

val partition : f:( 'a -> bool) -> 'a list -> 'a list * 'a list
  partition p l returns a pair of lists (l1, l2), where l1 is the list of all the elements of l that satisfy the predicate p, and l2 is the
  list of all the elements of l that do not satisfy p. The order of the elements in the input list is preserved.

```

Association lists

```
# List.map ~f:(+) 3 [4;5;6];;
```

```
# List.fold_left ~f:(+) ~init:0 [1;2;3;4;5];;
```

Optional Parameters

Optional Parameters

- Like labelled arguments, can be provided in any order
- Specify an optional value with the syntax
- `?(label = expresion)`

```
# let rec range ?(step=1) a b =  
  if a > b then []  
  else a :: range ~step (a + step) b;;  
val range : ?step:int -> int -> int list = <fun>
```

```
# range 1 10;;  
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

```
# range 1 10 ~step:2;;  
- : int list = [1; 3; 5; 7; 9]
```

- A function with option argument receives **None** when the caller doesn't provide the argument, and **Some** when it does.

```
# open Core.Std.String;;
```

```
# concat;;
```

```
- : ?sep:string -> string list -> string = <fun>
```

```
# concat ["foo";"bar"];;
```

```
- : string = "foobar"
```

```
# concat ?sep:None ["foo";"bar"];;
```

```
- : string = "foobar"
```

```
# let upper_concat ?sep a b = concat ?sep  
(List.map (uppercase 1));;
```

```
val upper_concat : ?sep:bytes -> bytes list -> bytes = <fun>
```


Optional arguments & partial application

```
# let foo ?(z = 0) x y = (x + y) > z;;  
val foo : ?z:int -> int -> int -> bool = <fun>
```

```
# let bar = foo 3;;  
val bar : int -> bool = <fun>
```

```
# bar 2;;  
- : bool = true
```

```
# bar 2 ~z:7;;
```

```
Error: This function has type int -> bool  
      It is applied to too many arguments;  
      maybe you forgot a `;'.
```

```
# let foo x ?(z = 0) y = (x + y) > z;;  
val foo : int -> ?z:int -> int -> bool = <fun>
```

```
# let bar = foo 3;;  
val bar : int -> bool = <fun>
```

```
# bar 2 ~z:7;;  
- : bool = false
```

Reference

- Labels (<https://ocaml.org/learn/tutorials/labels.html>)
- Real World OCaml (Chapter 2: Functions)