CSCI

Joosten

September 13th: More datastructures

University of Minnesota
Driven to Discover®

Outline

- Pairs
- Variant types again
- Lists again
- Some common types

Recursive code refresher

```
let rec takeWhilePositive lst =
  match lst with
  | [] -> []
  | hd :: tl ->
          if hd > 0 then hd :: takeWhilePositive tl
          else []
let rec dropWhilePositive lst =
  match lst with
  | [] -> []
  | hd :: tl ->
          if hd > 0 then dropWhilePositive tl
          else hd :: tl
```

or 'lst'

Pairs

```
let spanPositive lst
  = (takeWhilePositive lst, dropWhilePositive lst)
```

pair

```
val spanPositive :
  int list -> int list * int list = <fun>
```

Type of pair

Pairs of different types

pair

```
let foo = (2,[3;4])

val foo : ..?
```

Pairs of different types

pair

```
let foo = (2,[3;4])

val foo : int * int list = (2, [3; 4])
```

think (int list)

Lists of pairs

```
let bar = [2,3;4,5]

val bar : (int * int) list = [(2, 3); (4, 5)]
```

Rewriting our refresher …

```
let rec takeWhilePositive lst =
  match lst with
  | [] -> []
  | hd :: tl ->
        if hd > 0 then hd :: takeWhilePositive tl
        else []
let rec dropWhilePositive lst =
  match lst with
  | [] -> []
  | hd :: tl ->
        if hd > 0 then dropWhilePositive tl
        else hd :: tl
let spanPositive lst
  = (takeWhilePositive lst, dropWhilePositive lst)
```

Rewriting our refresher …

```
let rec spanPositive lst =
  match lst with
  | [] -> …
  | hd :: tl ->
      if hd > 0 then
      else

let spanPositive lst
  = (takeWhilePositive lst, dropWhilePositive lst)
```

Rewriting our refresher …

```
let rec spanPositive lst =
  match lst with
  | [] -> ([],[])
  | hd :: tl ->
      if hd > 0 then
      else …

let spanPositive lst
  = (takeWhilePositive lst, dropWhilePositive lst)
```

Rewriting our refresher …

```
let rec spanPositive lst =
  match lst with
  | [] -> ([],[])
  | hd :: tl ->
      if hd > 0 then …
      else ([], hd::tl)

let spanPositive lst
  = (takeWhilePositive lst, dropWhilePositive lst)
```

Rewriting our refresher …

```
let rec spanPositive lst =
  match lst with
  | [] -> ([],[])
  | hd :: tl ->
      if hd > 0 then (hd::tl1, tl2)
      else ([], hd::tl)

let spanPositive lst
  = (takeWhilePositive lst, dropWhilePositive lst)
```

Rewriting our refresher …

```
let rec spanPositive lst =
  match lst with
  | [] -> ([],[])
  | hd :: tl ->
      if hd > 0 then (let (tl1,tl2) = spanPositive tl
                       in (hd::tl1, tl2)
                     )
      else ([], hd::tl)

let spanPositive lst
  = (takeWhilePositive lst, dropWhilePositive lst)
```

Rewriting our refresher …

```
let rec spanPositive lst =
  match lst with
  | [] -> ([],[])
  | hd :: tl ->
      if hd > 0 then (match spanPositive tl with
                        (tl1,tl2) -> (hd::tl1, tl2)
                      )
      else ([], hd::tl)
```

this is the same thing,
but less conventional notation

```
let spanPositive lst
  = (takeWhilePositive lst, dropWhilePositive lst)
```

Rewriting our refresher …

```
let rec spanPositive lst =
  match lst with
  | [] -> ([],[])
  | hd :: tl ->
      if hd > 0 then (match spanPositive tl with
                        (tl1,tl2) -> (hd::tl1, tl2)
                     )
      else ([], hd::tl)

let takeWhilePositive lst
   = (let (res,_) = spanPositive lst in res)
let dropWhilePositive lst
   = (let (_,res) = spanPositive lst in res)
```

this is the same thing,
but less conventional notation

Variant types (again)

```
type operation = Times | Plus | Factorial
let exercise1 = (3, Times, 4)
let exercise2 = (5, Plus, 4)
let exercise3 = (2, Factorial)

val exercise1 : int * operation * int
 = (3, Times, 4)
val exercise2 : int * operation * int
 = (5, Plus, 4)
val exercise3 : int * operation
 = (2, Factorial)
```

Variant types (better)

```
type exercise
 = Times of (int * int)
 | Plus of (int * int)
 | Factorial of int
let exercise1 = Times (3,4)
let exercise2 = Plus (5,4)
let exercise3 = Factorial 2

val exercise1 : exercise = Times (3, 4)
val exercise2 : exercise = Plus (5, 4)
val exercise3 : exercise = Factorial 2

let exercises = [exercise1, exercise2, exercise3]
val exercises : (exercise * exercise * exercise) list
 = [(Times (3, 4), Plus (5, 4), Factorial 2)]
```

Using variant types …

```
type exercise
 = Times of (int * int)
 | Plus of (int * int)
 | Factorial of int

let string_of_exercise ex = match ex with
  | Times (x,y) -> string_of_int x ^ " * " ^
                   string_of_int y
  | Plus  (x,y) -> string_of_int x ^ " + " ^
                   string_of_int y
  | Factorial x -> string_of_int x ^ "!"
```

Lists again

```
type list_of_int
 = Empty
 | Cons of (int * list_of_int)
```

Recursive data type!

Lists again (better)

```
type 'a list
  = Empty
  | Cons of ('a * 'a list)
```

Polymorphic data type!
(and recursive)

Lists again (the way ocaml did it..)

```
type 'a list
  = []
  | (::) of 'a * 'a list

utop #
type 'a list = [] | (::) of 'a * 'a list;;
type 'a list = [] | (::) of 'a * 'a list
utop # let x = [1;2;3];;
val x : int list = (::) (1, (::) (2, (::) (3,
[])))
```

Note: the syntax sugar [1;2;3] now maps to our newly defined lists, but printing our lists doesn't use any syntax sugar yet.

Common data-types

- unit, bool
- option types

Common data-types

- unit and bool: types with 1 and 2 values

```
utop # ();;
- : unit = ()
utop # true;;
- : bool = true
utop # false;;
- : bool = false
```

```
type unit = ()
type bool = true | false
```
(informally)

Common data-types

- option types: value might be missing

```
utop # Some 3;;
- : int option = Some 3
utop # None;;
- : 'a option = None
```

Option types

```
utop # let divide_proper x y
         = if y = 0 then None
                    else Some (x/y);;
val divide_proper :
  int -> int -> int option = <fun>
```

What about composing computations?

Option types

```
utop # let divide_proper_twice x y z
          = if y = 0 || x = 0 then None
            else Some (z / (x/y));;
val divide_proper_twice :
  int -> int -> int -> int option = <fun>
```

This is highly error prone!
The point of divide_proper was to avoid /

Option types

```
utop # let divide_proper_twice x y z
         = if y = 0 || x = 0 then None
           else Some (z / (x/y));;
val divide_proper_twice :
  int -> int -> int -> int option = <fun>

utop # divide_proper_twice 3 4 5;;
Exception: Division_by_zero.
```

A cute helper function trick:

```
utop # let divide_proper x y
         = if y = 0 then None
                     else Some (x/y);;
val divide_proper :
  int -> int -> int option = <fun>
utop # let option_then (x : int option) f
         = match x with None -> None |
           Some y -> f y;;
val option_then :
  int option -> (int -> 'a option) ->
  'a option = <fun>
```

A cute helper function trick:

```
utop # let divide_proper x y
         = if y = 0 then None
                    else Some (x/y);;
val divide_proper :
  int -> int -> int option = <fun>
utop # let option_then (x : int option) f
         = match x with None -> None |
           Some y -> f y;;
utop # option_then (divide_proper 3 4)
                   (divide_proper 5)
- : int option = None
utop # option_then (divide_proper 4 2)
                   (divide_proper 5);;
- : int option = Some 2
```

Outlook

- The cute helper-function-trick for ‘option’ will come around more often, and more generally!
  (but much later in the course)
- I have yet to explain how type inference works