Programming Languages and Techniques (CIS120)

Lecture 12

September 27, 2017

Partiality, Sequencing, Records

Chapters 12, 13
Announcements

• Homework 3: Extension (due to Codio instabilities)
  – due TONIGHT at 11:59:59pm

• Homework 4
  – available now due on October 10th

• Midterm 1
  – October 13th in Class
  – Where? Last Names:
    A – M  Leidy Labs 10 (Here)
    N – Z  Meyerson Hall B1

  – Covers lecture material through Chapter 13
  – Review materials (old exams) on course website
  – Review session: TBA

• Dr. Sheth will be traveling 9/29 and 10/4 (Dr. Zdancewic will cover)
Dealing with Partiality*

*A function is said to be *partial* if it is not defined for all inputs.
Which of these is a function that calculates the maximum value in a (generic) list:

1. ```
   let rec list_max (l:'a list) : 'a = 
   begin match l with
   | [] -> []
   | h :: t -> max h (list_max t)
   end
``` 

2. ```
   let rec list_max (l:'a list) : 'a = 
   fold max 0 l
``` 

3. ```
   let rec list_max (l:'a list) : 'a = 
   begin match l with
   | h :: t -> max h (list_max t)
   end
``` 

4. None of the above

Answer: 4
• list_max isn’t defined for the empty list!

```ocaml
let rec list_max (l:'a list) : 'a =
begin match l with
    | [] -> failwith "empty list"
    | [h] -> h
    | h::t -> max h (list_max t)
end
```
Client of list_max

```haskell
(* string_of_max calls list_max *)
let string_of_max (x:int list) : string =
  string_of_int (list_max x)
```

- Oops! string_of_max will fail if given `[]`
- Not so easy to debug if string_of_max is written by one person and list_max is written by another.
- Interface of list_max is not very informative

```haskell
val list_max : int list -> int
```
Solutions to Partiality: Option 1

• Abort the program:
  `failwith “an error message”`
  – Whenever it is called, `failwith` halts the program and reports the error message it is given.

• This solution is appropriate whenever you know that a certain case is impossible
  – The compiler isn’t smart enough to figure out that the case is impossible...
  – Often happens when there is an invariant on a data structure
  – `failwith` is also useful to “stub out” unimplemented parts of your program.

• Languages (e.g. OCaml, Java) support exception handling facilities to let programs recover from such failures.
  – We'll talk about these when we get to Java
Solutions to Partiality: Option 2

- Return a *default or error value*
  - e.g. define `list_max []` to be -1
  - Error codes used often in C programs
  - `null` used often in Java

- But...
  - What if -1 (or whatever default you choose) really *is* the maximum value?
  - Can lead to many bugs if the default isn’t handled properly by the callers.
    - *IMPOSSIBLE* to implement generically!
      - No way to generically create a sensible default value for every possible type
      - Sir Tony Hoare, Turing Award winner and inventor of `null` calls it his “billion dollar mistake”!

- Defaults should be avoided if possible
Optional values

Solutions to Partiality: Option 3
Option Types

• Define a generic datatype of *optional values*:

  ```lean
  type 'a option =
  | None
  | Some of 'a
  ```

• A “partial” function returns an option

  ```lean
  let list_max (l:list) : int option = ...
  ```

• Contrast this with “null”, a “legal” return value of any type
  – caller can accidentally forget to check whether null was used; results in `NullReferenceException` or crashes

• Modern language designs (e.g. Apple's Swift, Mozilla's Rust) distinguish between the type `String` (definitely not null) and `String?` (optional string)
Example: list_max

- A function that returns the maximum value of a list as an option (None if the list is empty)

```ocaml
let list_max (l:'a list) : 'a option =
    begin match l with
      | [] -> None
      | x::tl -> Some (fold max x tl)
    end
```
Revised client of list_max

(* string_of_max calls list_max *)

let string_of_max (l:int list) : string =
  begin
    match (list_max l) with
    | None -> "no maximum"
    | Some m -> string_of_int m
  end

• string_of_max will never fail

• The type of list_max makes it explicit that a client must check for partiality.

  val list_max : int list -> int option
What is the type of this function?

```ocaml
define head (x: ______) : ______ =
begin
  match x with
  | []     -> None
  | h :: t  -> Some h
end
```

1. ‘a list -> ‘a
2. ‘a list -> ‘a list
3. ‘a list -> ‘b option
4. ‘a list -> ‘a option
4. None of the above

Answer: 4
What is the value of this expression?

```ocaml
let head (x: 'a list): 'a option =
    begin match x with
    | []    -> None
    | h :: t -> Some h
    end in

[ head [1];  head [] ]
```

1. [1; 0 ]
2. 1
3. [Some 1; None]
4. [None; None]
5. None of the above

Answer: 3
Revising the MAP interface

module type MAP = sig

    type ('k,'v) map

    val empty : ('k,'v) map
    val add : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
    val remove : 'k -> ('k,'v) map -> ('k,'v) map
    val mem : 'k -> ('k,'v) map -> bool
    val get : 'k -> ('k,'v) map -> 'v option
    val entries : ('k,'v) map -> ('k * 'v) list
    val equals : ('k,'v) map -> ('k,'v) map -> bool

end

get returns an optional 'v. Now its type isn't a lie!
Commands, Sequencing and Unit

What is the type of print_string?
Sequencing Commands and Expressions

We can *sequence* commands inside expressions using ‘;’

- unlike in C, Java, etc., ‘;’ doesn’t terminate a statement it *separates* a command from an expression

```plaintext
let f (x:int) : int =
  print_string "f called with ";
  print_string (string_of_int x);
  x + x
```

*do not use ‘;’ here!*

*note the use of ‘;’ here*

The distinction between commands & expressions is artificial.

- print_string is a function of type: string -> unit
- Commands are actually just expressions of type: unit
unit: the trivial type

• Similar to "void" in Java or C

• For functions that don't take any arguments

```ocaml
let f () : int = 3
let y : int = f ()
```

• Also for functions that don't return anything, such as testing and printing functions a.k.a commands:

```ocaml
(val f : unit -> int
  val y : int

(* run_test : string -> (unit -> bool) -> unit *)
;; run_test "TestName" test

(* print_string : string -> unit *)
;; print_string "Hello, world!"
```
unit: the boring type

- Actually, () is a value just like any other value (a 0-ary tuple)
- For functions that don't take any interesting arguments

```
let f () : int = 3
let y : int = f ()
```

- Also for functions that don't return anything interesting, such as testing and printing functions a.k.a commands:

```
(* run_test : string -> (unit -> bool) -> unit *)
;; run_test "TestName" test

(* print_string : string -> unit *)
;; print_string "Hello, world!"
```
unit: the first-class type

- Can define values of type `unit`
  ```
  let x : unit = ()
  val x : unit
  ```

- Can pattern match `unit` (even in function definitions)
  ```
  let z = begin match x with
       | () -> 4
   end
  fun () -> 3
  ```

- Is the result of an implicit else branch:
  ```
  ;; if z <> 4 then
  failwith "oops"
  ;; if z <> 4 then
  failwith "oops"
  else ()
  ```
Sequencing Commands and Expressions

• Expressions of type unit are useful because of their side effects – they "do" stuff
  — e.g. printing, changing the value of mutable state

```
let f (x:int) : int =
    print_string "f called with ";
    print_string (string_of_int x);
    x + x
```

• We can think of ‘;’ as an infix function of type:
  unit -> ‘a -> ‘a

CIS120
What is the type of $f$ in the following program:

```
let f (x:int) =
  print_int (x + x)
```

1. unit -> int
2. unit -> unit
3. int -> unit
4. int -> int
5. $f$ is ill typed

Answer: 3
What is the type of f in the following program:

```
let f (x:int) = 
  (print_int x);
  (x + x)
```

1. unit -> int
2. unit -> unit
3. int -> unit
4. int -> int
5. f is ill typed

Answer: 4
Records
Immutable Records

• Records are like tuples with named fields:

(* a type for representing colors *)

```plaintext
type rgb = {r:int; g:int; b:int;}
```

(* some example rgb values *)

```plaintext
let red : rgb = {r=255; g=0; b=0;}
let blue : rgb = {r=0; g=0; b=255;}
let green : rgb = {r=0; g=255; b=0;}
let black : rgb = {r=0; g=0; b=0;}
let white : rgb = {r=255; g=255; b=255;}
```

• The type rgb is a record with three fields: r, g, and b
  – fields can have any types; they don’t all have to be the same

• Record values are created using this notation:
  `{field1=val1; field2=val2;…}`
Field Projection

- The value in a record field can be obtained by using “dot” notation: `record.field`

```plaintext
(* a type for representing colors *)
type rgb = {r:int; g:int; b:int;}

(* using 'dot' notation to project out components *)
(* calculate the average of two colors *)
let average_rgb (c1:rgb) (c2:rgb) : rgb =
  {r = (c1.r + c2.r) / 2;
   g = (c1.g + c2.g) / 2;
   b = (c1.b + c2.b) / 2;}
```
OCaml provides convenient syntax for working with records:

```ocaml
let f {r ; g} = r + g

let mk_rgb (r:int) (g:int) (b:int) = {r; g; b}
```