Programming Languages and Techniques (CIS120)

Lecture 12

September 23rd , 2015

Finite Maps; Partiality and Option Types

Announcements

- ## Homework 3
  - Due tomorrow *Thursday September 24th* at 11:59:59pm
  - next homework will be available Monday or Tues.

- ## Midterm 1
  - In class on *Friday, October 2nd*
    - Last names   A – L   Leidy Labs (here)
    - Last names   M – Z   Cohen G17

  - Covers lecture material through TODAY
  - Review materials (old exams) on course website
  - Contact me if you need to take the make-up exam

Finite Map *signature*

```
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
  val entries : ('k,'v) map -> ('k * 'v) list
  val equals  : ('k,'v) map -> ('k,'v) map -> bool

end
```

Summary: Abstract Types

- Different programming languages have different ways of letting you define abstract types

- At a minimum, this means providing:
  - A way to specify (write down) an interface
  - A means of hiding implementation details (*encapsulation*)

- In OCaml:
  - Interfaces are specified using a *signature* or *interface*
  - Encapsulation is achieved because the interface can *omit* information
    - type definitions
    - names and types of auxiliary functions
  - Clients *cannot* mention values or types not named in the interface

Which of these is a function that calculates the maximum value in a 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

Quiz answer

- list_max isn't defined for the empty list!

```
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
```

```
(* INCORRECT! *)
let list_max (l:'a list) : 'a =
  begin match l with
    | [] -> failwith "empty list"
    | h::t -> fold max h t
  end
```

Client of list_max

```
(* 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
  ```
  val list_max : int list -> int
  ```

Dealing with Partiality*

*A function is said to be *partial* if it is not defined for all inputs.

Solutions to Partiality: Option 1

- Abort the program:
  - In OCaml: `failwith "an error message"`

- Whenever it is called, `failwith` aborts the program and reports the error message it is given.

- This solution to partiality is appropriate whenever you *know* that a certain case is impossible.
  - Often happens when there is an invariant on a datastructure
  - The compiler isn't smart enough to figure out that the case is impossible…
  - `failwith` is also useful to "stub out" unimplemented parts of your program.

- This is an instance of a more general mechanism called "exceptions"

Solutions to Partiality: Option 2

- Return a *default or error value*
  - e.g. define `tree_max Empty` 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 or error value isn't handled properly by the callers.

  - *IMPOSSIBLE* to implement generically!
    - There is 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*:

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

- A "partial" function returns an option

```
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 NullPointerExceptions or crashes

Example: list_max

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

```
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?

```
let 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?

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

head [[1]]
```

1. 1

2. Some 1

3. [1]

4. Some [1]

5. None of the above

Answer: 4

What is the value of this expression?

```
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

Revisiting the MAP interface

get : 'k -> ('k,'v) map -> 'v option