Programming Languages and Techniques (CIS120)

Lecture 12

Feb 8, 2012

Options, Unit and (Mutable!) Records

Announcements

- Homework 4 is available on the web
  - due Monday, February 13th at 11:59:59pm
  - n-body physics simulation
  - start early; see Piazza for discussions

- Updated lecture notes also available…
  - New language features in homework 4

- Midterm 1 will be in class on Wednesday, February 15th
  - LOCATION: LLAB 10
  - Review materials on website
  - Bring questions to lab
  - Review session Tuesday evening
  - Let me know about scheduling problems ASAP

Quick quiz

- Write a recursive function to calculate the maximum value in a list of numbers

```
let rec list_max (l:'a list) : ’a =
```

Quiz answer

- Write a recursive function to calculate the maximum value in a list of numbers

```
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
```

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

Dealing with Partiality

Option Types

Partial Functions

- Sometimes functions aren't defined for all inputs:
  - tree_max from the BST implementation isn't defined for empty trees
  - integer division by 0
  - Map.find k m when the key k isn't in the finite map m
- We have seen how to deal with partiality using failwith
  - but failwith aborts the program
- Can we do better?
- Hint: we already have all the technology we need.

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 value, a "legal" return value of any type
  - caller can accidentally forget to check whether null was used; results in NullPointerExceptions or crashes
  - Sir Tony Hoare, Turing Award winner and inventor of "null" calls it his "*billion dollar mistake*"!

Example: list_max

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

```
let rec list_max (l:int list) : int option =
  begin match l with
    | [] -> None
    | x::tl -> begin match (list_max tl) with
                 | None -> Some x
                 | Some y -> if x > y
                             then Some x
                             else Some y
               end
  end
```

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
```

Unit

unit: the trivial type

- Similar to "void" in Java or C
- For functions that don't take any arguments

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

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

- Also for functions that don't return anything, 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 boring type

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

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

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

- 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 = ()
```

```
val x : unit
```

- Can pattern match unit (even in function definitions)

```
let z = begin match x with
  | () -> 4
end
```

```
fun () -> 3
```

- Is the implicit else branch:

```
;; if z <> 4 then
    failwith "test failed"
```

=

```
;; if z <> 4 then
    failwith "test failed"
  else ()
```

Sequencing Commands and Expressions

- Expressions of type unit are useful because of their *side effects* (e.g. printing)
- We can *sequence* those effects using ‘;’
  - unlike in C, Java, etc., ‘;’ doesn’t terminate a statement it *separates* a command from an expression

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

do *not* use ‘;’ here!

note the use of ‘;’ here

- We can think of ‘;’ as an infix function of type:

```
unit -> 'a -> 'a
```

Records

Records

- Records are like tuples with named fields:

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

(* some example rgb values *)
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`

```
(* 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;}
```

“with” notation for *copying* records

- Sometimes it is useful to *copy* one record while replacing just a few of its fields:

```
(* using 'with' notation to copy a record but
   change one (or more) fields *)
let cyan = {blue with g=255}
let magenta = {red with b=255}
let yellow = {green with r=255}
```

- Syntax: `{record with field1=val1; field2=val2}`
  – note the ‘{‘ and ‘}’

Imperative Programming

Course Overview

- Declarative programming
  - *persistent* data structures
  - *recursion* is main control structure
  - heavy use of functions as data

We are here.
Midterm 1 covers
material up to this point.

- Imperative programming
  - *mutable* data structures (that can be modified "in place")
  - *iteration* is main control structure
- Object-oriented programming
  - pervasive "abstraction by default"
  - mutable data structures / iteration
  - heavy use of functions (objects) as data

Why Use Declarative Programming?

- Simple
  - small language: arithmetic, local variables, recursive functions, datatypes, pattern matching, polymorphism and modules
  - simple substitution model of computation
- Persistent data structures
  - Nothing changes, so can remember all intermediate results
  - Good for version control, fault tolerance, etc.
- Typecheckers give more helpful errors
  - Once your program compiles, it needs less testing
  - failwith vs. NullPointerException
- Easier to parallelize and distribute
  - No implicit interactions between parts of the program. All of the behavior of a function is specified by its arguments

Why Use Mutable State?

- Action at a distance
  - allow remote parts of a program to communicate / share information without threading the information through all the points in between
- Direct manipulation of hardware (device drivers, etc.)
- Data structures with explicit sharing
  - e.g. graphs
  - without mutation, it is only possible to build trees – no cycles
- Efficiency/Performance
  - a few data structures have imperative versions with better asymptotic efficiency than the best declarative version
- Re-using space (in-place update)
- Random-access data (arrays)

A new view of imperative programming

Java (and C, C++, C#)

- Null is contained in (almost) every type. Partial functions can return **null**.
- Code is a sequence of **statements** that do something, sometimes using expressions to compute values.
- References are **mutable** by default, must be explicitly declared to be constant

OCaml

- No null. Partiality must be made explicit with **options**.
- Code is an **expression** that has a value. Sometimes computing that value has other effects.
- References are **immutable** by default, must be explicitly declared to be mutable

Mutable Record Fields

- By default, all record fields are *immutable*—once initialized, they can never be modified.
- OCaml supports *mutable* fields that can be imperatively updated by the "set" command: `record.field <- val`

note the 'mutable' keyword

```
type point = {mutable x:int; mutable y:int}

let p0 = {x=0; y=0}
(* set the x coord of p0 to 17 *)
;; p0.x <- 17
;; print_endline ("p0.x = " ^ (string_of_int p0.x))
```

"in-place" update of p0.x

Defining new Commands

- Functions can assign to mutable record fields
- Note that the return type of '<-' is unit

```
type point = {mutable x:int; mutable y:int}

(* a command to shift a point by dx,dy *)
let shift (p:point) (dx:int) (dy:int) : unit =
  p.x <- p.x + dx;
  p.y <- p.y + dy
```