Programming Languages and Techniques (CIS120)

Lecture 10

Feb 3, 2012

First-class functions

Announcements

- Homework 3 is due Monday at 11:59:59pm
- Midterm 1 will be in class on Wednesday, February 15^{th}

Finite Map Demo

Using module signatures to preserve data structure invariants

Finite Maps

- A *finite map* is a collection of *bindings* from distinct *keys* to *values*.
  - Operations to add & remove bindings, test for key membership, lookup a value by its key

- Example: an `(ID, int) map` might map a PennKey ID to the lab section.

- Like sets, such finite maps appear in many settings:
  - map domain names to IP addresses
  - map words to their definitions (a dictionary)
  - map user names to passwords
  - map game character unique identifiers to dialog trees
  - …

Demo: Map.ml

Abstracting with first-class functions

Finite Map Interface

```
type ('k,'v) map

val empty    : ('k,'v) map
val is_empty : ('k,'v) map -> bool
val mem      : 'k -> ('k,'v) map -> bool
val find     : 'k -> ('k,'v) map -> 'v
val add      : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
val remove   : 'k -> ('k,'v) map -> ('k,'v) map

val from_list : (‘k * ‘v) list -> (‘k,’v) map
val bindings  : (‘k,’v) map -> (‘k * ‘v) list
```

Motivating design problem

- Suppose you are given a finite map from students to majors, but you wanted a map that includes only students in the engineering school? Or only students in wharton?

```
type student = string
type major   = string
type school  = SEAS | WHARTON | SAS | NURSING
type roster = (student,major) map

let to_school (m : major) : school = ...

let is_engr (m : major) : bool = to_school m = SEAS
let is_wharton (m : major) : bool = to_school m = WHARTON

let only_engr (r : roster) : roster = ???
let only_wharton (r : roster) : roster = ???
```

Demo: Majors.ml

First-class Functions

- Amazing fact: functions *are* data!
- You can pass a function as an *argument* to another function:

```
let twice (f:int -> int) (x:int) : int =
   f (f x)

let add_one (z:int) : int = z + 1
```

- You can *return* a function as the result of another function.

```
let make_incr (n:int) : int -> int =
  let helper (x:int) : int =
        n + x
  in
  helper
```

Evaluating First-Class Functions

```
let twice (f:int -> int) (x:int) : int =
   f (f x)

let add_one (z:int) : int = z + 1
```

	`twice add_one 3`	
$\longmapsto$	`add_one (add_one 3)`	*substitute add_one for f, 3 for x*
$\longmapsto$	`add_one (3 + 1)`	*substitute 3 for z in add_one*
$\longmapsto$	`add_one 4`	*because 3+1⇒4*
$\longmapsto$	`4 + 1`	*substitute 4 for z in add_one*
$\longmapsto$	`5`	*because 4+1⇒5*