CS 3110

Lecture 10: Functors

Prof. Clarkson

Fall 2014

Today's music: "Nice to know you" by Incubus
"...It's hard for me to specify..."

Review

First month of course:

- Programming in the small
  - Lots of language features
  - Lots of small functions

This week:

- Programming in the large
  - A few new language features (modules, signatures)
  - Modularity, abstraction
- Today:
  - Specification
  - Functors

Question #1

Think about `java.util` (or some other library you've used frequently). How do you usually come to understand the functionality it provides?

A. **By example:** I search until I find code using the library, then tweak the code to do what I want.
B. **By tutorial:** I read the library's tutorial to understand how it works, then I write code inspired by it.
C. **By documentation:** I read the official documentation for functions, classes, etc., in the library, then I write code from scratch.
D. **By implementation:** I download the source code for the library, read it, then write my own code.
E. I never really understood `java.util`.

ABSTRACT TYPES

Review: stack with abstract types

```
module type STACK = sig
    type 'a t
    val empty : 'a t
    val is_empty : 'a t -> bool
    val push : 'a -> 'a t -> 'a t
    val pop : 'a t -> 'a * 'a t
end

module Stack : STACK = struct
    type 'a t = 'a list
    let empty = []
    let is_empty s = s = []
    let push x s = x :: s
    let pop s = match s with
      [] -> failwith "Empty"
    | x::xs -> (x,xs)
end
```

Recall: procedural and data abstraction

Abstract type inside stack

Why hide the fact that a stack is an ?

General principle: information hiding

- *Clients* of Stack don't need to know it's implemented with a list
- *Implementers* of Stack might one day want to change the implementation
  - If list implementation is exposed, they can't without breaking all their clients' code
  - If list implementation is hidden, they can freely change

Example?

- Honestly, hard with the **`Stack`** signature we have
- Many languages simply supply **`pop`** and **`push`** functions for lists
- But suppose we want to support a **`min`** function...

Stacks with min

```
module type STACK = sig
    type 'a t
    val empty : 'a t
    val is_empty : 'a t -> bool
    val push : 'a -> 'a t -> 'a t
    val pop : 'a t -> 'a * 'a t
    val min : 'a t -> 'a option
end
```

Stacks with min

```
module Stack : STACK = struct
    type 'a t = 'a list
    let empty = []
    let is_empty s = s = []
    let push x s = x :: s
    let pop s = match s with
         [] -> failwith "Empty"
       | x::xs -> (x,xs)
    let min s = list_min s
end
```

Suppose we want to support O(1) **`min`**, and are okay with more expensive **`pop`**

Reimplemented stack

```
module StackEffMin : STACK = struct
    (* In S(m,lst), the list must never be empty,
        and m must be the minimum value in the stack *)
    type 'a t = Empty | S of 'a * 'a list
    let is_empty ms = ms = Empty
    let push x ms =
        match ms with
            Empty -> S (x,[x])
        | S(m,s) -> S (min x m, x :: s)
    let min ms =
        match ms with
            Empty -> None
        | S(m,_) -> Some m
    ...
    (* pop is more expensive *)
end
```

Reimplemented stack

- The *representation type* changed
  - from `'a list`
  - to `Empty | S of 'a * 'a list`
- If type is abstract in signature, clients continue to compile
- If type is revealed in signature, clients who relied on a list fail to compile
- For more complicated data structures, this problem just gets worse
  - e.g., suppose Microsoft wants to update the data structure representing a window or canvas or file or…

Other data structures

- In recitation: stacks, queues, dictionaries, fractions
- All are *functional data structures:*
  - never destructively update the data structure
  - instead, apply functions that produce a new copy of the data structure with some changes applied
  - both copies are still available for use

Set data structure

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val mem : 'a -> 'a set -> bool
  val add : 'a -> 'a set -> 'a set
  val size: 'a set -> int
end

module ListSet : SET = struct
  (* the list may never have duplicates *)
  type 'a set = 'a list
  let empty = []
  let mem = List.mem
  let add x l = if mem x l then l else x :: l
  let size = List.length
end
```

Set data structures

How does **`List.mem`** check for membership?

```
let rec mem x = function
    [] -> false
  | a::l -> compare a x = 0 || mem x l
```

What is **`compare`**?

"compare x y returns 0 if x is equal to y, a negative integer if x is less than y, and a positive integer if x is greater than y." [Pervasives.mli]

How does **`compare`** work?

- Abstraction: spec doesn't say
- Implementation calls into C code [e.g., byterun/str.c]

Set data structures

- Suppose we want a set with a relaxed notion of equality
  - Case-insensitive strings
  - + or – insensitive ints
- Ideas???

Question #2

How would you design a set abstraction that allows relaxed notions of equality?

A. Ask client to preprocess each item as added to set
B. Ask client to pass in a customized comparison function as argument to each set function
C. Store a comparison function as part of the representation type of the set
D. Something else...

Set data structures

- Could ask client to preprocess each item as added to set
  - But client might forget
- Could pass in a customized comparison function
  - But client has to pass it in everytime mem or add is called
- Could store function as part of representation type
  - But no longer possible to tell from type of set what kind of comparison it will use
- Probably many other ideas… OCaml has a great feature called **functors** that is designed to help

Functor

A **functor** is a “function” from modules to modules

- Module-level functions
- Written with different syntax than value-level functions
- Have functor types, written with different syntax than value-level function types

Simple functor

```
module type XINT = sig
   val x : int
end
module Three : XINT = struct
   let x = 3
end

module IncFn(M:XINT) : XINT = struct
   let x = M.x+1
end
module Four = IncFn(Three)

Four.x - Three.x --> 1
```

Alternative syntax

```
module IncFn(M:XINT):XINT = struct
  let x = M.x+1
end
(* or *)
module IncFn =
  functor (M: XINT) ->
  (struct
    let x = M.x+1
  end : XINT)
(* cannot write "return type"
 * to the left of arrow *)
```

A nifty functor trick

Can write a functor to do the following:

- Take any module that contains **`fold`** function
- Produce a new module that contains everything implementable with just **`fold`**
  - **`iter, length, for_all,`** etc.
- Functions for free!
  - see chap. 9 of Real World OCaml
  - Ruby has a similar idiom with **`Enumerable`**
    - (write an iterator **`each`**, get many functions for free)

But back to sets...

Equality signature

```
module type EQUAL = sig
   type t
   val equal : t -> t -> bool
end
module StringEqual : EQUAL = struct
   type t = string
   let equal = (=)
end
module StringCaseInsEqual : EQUAL = struct
   type t = string
   let equal s t =
     String.uppercase s = String.uppercase t
end
```

Using equality modules

```
# StringCaseInsEqual.equals "s" "S"
```

```
Error: This expression has type
string but an expression was
expected of type
StringCaseInsAbsTypeEqual.t
```

Problem: outside module, nobody knows what `t` is, so can't pass in strings!

Solution: expose the abstract type

Type exposure

```
module StringCaseInsEqual :
   (EQUAL with type t = string) =
struct
   type t = string
   let equal s t =
      String.uppercase s = String.uppercase t
end
```

Sharing constraint: shares with outside world what abstract type really is

Set functor

```
module MakeSetFn (Equal: EQUAL) = struct
  type elt = Equal.t
  (* the list may never have duplicates *)
  type set = elt list
  let empty = []
  let mem x = List.exists (Equal.equal x)
  let add x l = if mem x l then l else x :: l
  let size = List.length
end

module StringSet = MakeSetFn(StringEqual)
module CaseInsStringSet =
  MakeSetFn(StringCaseInsEqual)
```

Type of set functor?

```
module type SET_FN =
  functor (Equal : EQUAL) -> sig
    type elt = Equal.t
    type set
    val empty : set
    val mem : elt -> set -> bool
    val add : elt -> set -> set
    val size: set -> int
  end
module MakeSetFn : SET_FN =
  functor (Equal: EQUAL) -> struct
    (* as on previous slide ... *)
  end
```

ABSTRACTION

Abstraction techniques

Procedural and data abstraction share two common techniques:

- Abstraction by parameterization
- Abstraction by specification

Abstraction by parameterization

- Introduce parameters to functions
- Use those parameters instead of hardcoded values, e.g.,
  - instead of `a*a+b*b`,
  - write `let sum_squares x y -> x*x + y*y`,
  - and call `sum_squares a b`
- you basically take abstraction by parameterization for granted in any modern language

Abstraction by specification

- Document behavior of function
  - Primarily, with pre- and postconditions
  - Use documentation to reason about behavior
    - instead of having to read implementation

- We've been teaching you this for three semesters now, I hope...but...
  - the language syntax doesn't demand it
  - the compiler doesn't checks it
  - ...so writing good specs is a skill that takes longer to mature

Example specification

```
val sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

Sort a list in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see Array.sort for a complete specification). For example, `compare` is a suitable comparison function. The resulting list is sorted in increasing order. `List.sort` is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.

Exercise: take 2 minutes. Feel free to talk with someone near you. Identify any preconditions and postconditions.

Example specification

- Sort a list in increasing order according to a comparison function.
- The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see Array.sort for a complete specification). For example, compare is a suitable comparison function.
- The resulting list is sorted in increasing order.
- List.sort is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.

Example specification

- **One-line summary of behavior:** Sort a list in increasing order according to a comparison function.
- **Precondition:** The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see Array.sort for a complete specification). For example, compare is a suitable comparison function.
- **Postcondition:** The resulting list is sorted in increasing order.
- **Promise about behavior:** List.sort is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.

Question #3

What grade would you give the List.sort specification?

A. It provides pre- and postconditions. They are specific enough for me to understand how to use the function as a client. They do not contain irrelevant details or vague descriptions.
B. Parts of the specification are hard to understand. Some details are missing, or some parts are vague.
C. The specification is confusing or just plain wrong.

What if you had to read the implementation?

list.ml

```
      let l2 = chop n1 l in
      let s1 = rev_sort n1 l in
      let s2 = rev_sort n2 l2 in
      rev_merge_rev s1 s2 []
  and rev_sort n l =
    match n, l with
    | 2, x1 :: x2 :: _ ->
      if cmp x1 x2 > 0 then [x1; x2] else [x2; x1]
    | 3, x1 :: x2 :: x3 :: _ ->
      if cmp x1 x2 > 0 then begin
        if cmp x2 x3 > 0 then [x1; x2; x3]
        else if cmp x1 x3 > 0 then [x1; x3; x2]
        else [x3; x1; x2]
      end else begin
        if cmp x1 x3 > 0 then [x2; x1; x3]
        else if cmp x2 x3 > 0 then [x2; x3; x1]
        else [x3; x2; x1]
      end
    | n, l ->
      let n1 = n asr 1 in
      let n2 = n - n1 in
      let l2 = chop n1 l in
      let s1 = sort n1 l in
      let s2 = sort n2 l2 in
      rev_merge s1 s2 []
  in
  let len = length l in
  if len < 2 then l else sort len l
;;

let sort = stable_sort;;
let fast_sort = stable_sort;;
```

-:--- list.ml   58% L263   (Tuareg merlin (default))

Please hold still for 1 more minute

WRAP-UP FOR TODAY

Upcoming events

- **PS3 due in one week**
- Clarkson's office hours today as usual

This is abstract.

THIS IS 3110