Overview

Problem Set 2 consists of this writeup as well as ps2.ml and ps2.mli, which contain the release code for this assignment. The release code contains the following types, as well as signatures and skeleton implementations for the modules and functions you are to implement. These skeleton implementations consist of failwiths that will allow you to compile and use the functions and modules you have implemented without having to implement everything else. Recall that OCaml will type-check your implementations when you compile them using cs3110 compile, helping you catch some errors before ever running your code. You should take advantage of this while working on the assignment to avoid having to debug everything at once.

In particular, if your code compiles with the provided ps2.mli, then your implementations conform to the types we have specified—which is very important. Code that does not compile with cs3110 compile and the provided signature will be heavily penalized.

The following types are defined at the top of ps2.ml and ps2.mli. They are referenced in later problems.

```ocaml
type cmp =
  | Less
  | Greater
  | Equal

type ('a, 'b) part =
  | Left of 'a
  | Right of 'b
  | Discard
```

Folding and Recursion

For this problem, write two implementations for each of the following functions: one should use recursion and the other should use either List.fold_left or List.fold_right. You can perform post-processing on the result of the fold.

1. map : ('a -> 'b) -> 'a list -> 'b list
   map f [a1; ...; an] is [f a1; ... f an]. The order of elements must be preserved.
2. `partition : ('a -> ('b, 'c) part) -> 'a list -> 'b list * 'c list`

`partition f a` returns a pair of lists `b` and `c`, where each element `\beta` of `b` corresponds to an element `\alpha` of `a` such that `f \alpha = \text{Left } \beta`, and where each element `\gamma` of `c` corresponds to an element `\alpha` of `a` such that `f \alpha = \text{Right } \gamma`. Each element `\alpha` of `a` such that `f \alpha = \text{Discard}` does not correspond to any element in `b` or `c`.

3. `mean : float list -> float`

`mean [a1; ...; an]` returns the arithmetic mean of `a1, ..., an`.

4. `all_prefixes : 'a list -> 'a list list`

`all_prefixes ls` returns a list of all prefixes of `ls`, including the empty list, ordered from longest prefix to shortest prefix. For example, `all_prefixes [1; 2; 3]` is `[[1; 2; 3]; [1; 2]; [1]; []].`

5. `average_heading : float list -> float`

`average_heading [h1; ...; hn]` takes floats `h1` through `hn` in a list, where each `hi` indicates a compass heading between 0 and 360 inclusive, and returns the average of their headings as a compass heading. Note that the average of 0 and 360 is not 180—it is 360 (equivalently, 0). Behavior when an average heading cannot be well defined (for example, what is the average of the headings 0 and 180?) is undefined.

**Natural Numbers**

_Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk._

God made the integers, all else is the work of man.

---

Leopold Kronecker

In 1889, the Italian mathematician Giuseppe Peano published an axiomatic definition of the natural numbers. Here is the gist of it:

(i) 0 is a natural number, and

(ii) if `n` is a natural number, `S(n)` is a natural number.

Interestingly, although at first this definition might seem much more theoretical than practical, it turns out that this definition of the natural numbers can sometimes come in
handy for “practical” purposes. For example, the Sized Linear Algebra Package\(^1\) library for OCaml implements static size-checking for matrix operations by encoding the sizes of matrices in their types as follows:

```ocaml
1 type z
2 type 'n s

... so z s s represents 0 + 1 + 1.
```

**Modules for the natural numbers**

For this problem, implement two modules for the natural numbers: one using ints, and the other using Peano’s definition of the natural numbers. Both should implement the following signature:

```ocaml
module type N = sig

(** [t] represents a natural number. *)

type t

(** [zero] returns the [t] representing 0. *)
val zero : t

(** [succ x] returns the successor to [x]. *)
val succ : t -> t

(** [eq a b] returns [a = b] (but you might implement it differently! *)
val eq : t -> t -> bool

end
```

The first module should be called **IntNat** and implement \(N\) using ints. The second module should be called **PeanoNat** and implement \(N\) without using ints.

Note the following important things:

(i) ★ This signature makes the type \(t\) abstract. You should not use facts about your implementation of \(t\) for the following problems.

(ii) ★ OCaml provides a structural comparison operator, =. Even though your implementation will likely allow = to return true iff two ts are equal, because = breaks the abstraction barrier, you should only use eq to compare two ts.

\(^1\)https://github.com/akabe/slap
(iii) **zero** is a value, not a function returning 0. It’s immutable, which is why it’s safe to expose it like this.

(iv) Following from (i), the only way to obtain an N.t other than **zero** outside of the module other than is through **succ**.

### 2.2 Operations on the natural numbers

Here is the signature for a module implementing some operations on the natural numbers:

```ocaml
module type NAT_OPS = sig
  (* We expose that [t = PeanoNat.t] so that [PeanoNat.t]s can be used with the functions in this module. *)
  type t = PeanoNat.t

  (** [add a b] returns [a + b]. *)
  val add : t -> t -> t

  (** [sub a b] returns [a - b]. If [a < b], the result is undefined. *)
  val sub : t -> t -> t

  (** [cmp a b] returns [Less] if [a < b], [Greater] if [a > b], and [Equal] if [a = b]. *)
  val cmp : t -> t -> cmp

  (** [mul a b] returns [a * b]. *)
  val mul : t -> t -> t

  (** [div a b] returns [(q, r)] such that [a = bq + r] with [r < b]. *)
  val div : t -> t -> t * t

end
```

For this problem, implement a module **NatOps** implementing **NAT_OPS**, with **NatOps.t** equal to **PeanoNat.t**.

Note that you are implementing these functions in a separate module outside of **PeanoNat** to enforce the abstractness of the type **PeanoNat.t**.
Rational Numbers

A rational number is a number of the form \( \frac{p}{q} \), where \( p \) and \( q \) are integers and \( q \) is nonzero. 2/1, 4/2, and 3/7 are rational numbers; \( \pi \) and \( e \) are not.

All natural numbers are integers, all integers are rational numbers, all rational numbers are real numbers, and so on. Some languages (in particular Scheme and some Lisps) support a numeric tower for the representation of numbers, where the data types for numbers are implemented as a “tower” of more complicated types for more complicated numbers, all which can easily interact with each other in predictable ways. OCaml does not implement a numeric tower, but does provide ints, floats, and even arbitrary-precision Big_int.ts. It does not provide a type for rational numbers specifically.

For this problem, implement a module **Rational** for the rational numbers with the following signature:

```ocaml
module type Q = sig
  type t
  val make : int -> int -> t option
  val add : t -> t -> t
  val inv : t -> t
  val neg : t -> t
  val mul : t -> t -> t
  val eq : t -> t -> bool
  val cmp : t -> t -> cmp

let r1 = make 2 3
let r2 = make 1 3
let r3 = make 3 2
let r4 = make 3 1
let r5 = make 1 2
let r6 = make 2 1

let r7 = inv r1
let r8 = inv r2
let r9 = inv r3
let r10 = inv r4
let r11 = inv r5
let r12 = inv r6

let r13 = neg r1
let r14 = neg r2
let r15 = neg r3
let r16 = neg r4
let r17 = neg r5
let r18 = neg r6

let r19 = mul r1 r2
let r20 = mul r2 r3
let r21 = mul r3 r4
let r22 = mul r4 r5
let r23 = mul r5 r6
let r24 = mul r6 r7

let r25 = eq r1 r2
let r26 = eq r2 r3
let r27 = eq r3 r4
let r28 = eq r4 r5
let r29 = eq r5 r6
let r30 = eq r6 r7

let r31 = cmp r1 r2
let r32 = cmp r2 r3
let r33 = cmp r3 r4
let r34 = cmp r4 r5
let r35 = cmp r5 r6
let r36 = cmp r6 r7
```

26 (** [div a b] returns [a / b]. *)
27 val div : t -> t -> t
28 
29 (** [gcd a b] returns the gcd of [a] and [b]. *)
30 val gcd : t -> t -> t
31 
32 (** [to_int x] returns [p] and [q] such that [p/q = x]. *)
33 val to_int : t -> int * int
34 
35 end

As an additional resource, please refer to the notes for lecture 5 at http://www.cs.cornell.edu/courses/cs3110/2016sp/lectures/5/CS3110-sp16-Lec5.pdf.

Future Problem

IMPORTANT NOTICE

There is additional material in this problem set that will be released not later than Monday, February 21.

The release of the additional material will be announced in lecture and on Piazza.

Thanks!

IMPORTANT NOTICE
Skip Sets

A skip list is a probabilistic data structure that allows for average $O(\lg n)$ search, insert, and delete on an ordered sequence of keys, each associated with a value. It is essentially a list of lists emulating a binary search tree.

For example, the following figure represents a skip list containing 10 key-value pairs numbered 1 through 10. Each box is a cons cell of the form $x :: y$, where the box contains $x$ and the arrow points to $y$. Each row represents one of the component lists. The NILs correspond to $[]$ in OCaml.

Let’s suppose we’re searching for an element $x$ and that the sequence is sorted in increasing order. We start at the top-most list and go right until the current element is either greater than $x$ or is $[]$. Then we go back to the previous element (which less than $x$) and jump one level down to the list below. We proceed this way until we reach the bottom-most level, below which are the elements we are searching for. If you trace a path like this on the diagram, you should see that we take on average $O(\lg n)$ steps. If the upper-level lists are particularly badly chosen, we can be forced to take up to $O(n)$ steps.

For this problem, implement a skip set, a simplified version of a skip list only supporting offline creation and querying. Your module should be called SkipSet and implement the following signature:

```ocaml
module type SKIP_SET = sig

(** An ['a t] represents a set containing elements of type ['a]. *)
val type 'a t

(** [make p xs] creates a new [t] containing the elements of [xs]. [p] is the probability that an element in a level of the skip set is present in the next level of the skip set. *)
val make : float -> 'a list -> 'a t

end
```
(* [mem x set] returns [true] iff [x] is an element of [set]. *)

val mem : 'a -> 'a t -> bool

end

Our first-pass implementation is about three times as slow as OCaml’s Set on floats with $p = 0.25$ (but still more than five hundred times faster than a brute-force search). Try to make yours faster! At the least, your implementation of mem should not be slower than a brute-force implementation.

For further reference, the Wikipedia article at https://en.wikipedia.org/wiki/Skip_list might be helpful.

Comments

In the provided written.txt file, please include any comments you have about the problem set or your implementation. This would be a good place to list any problems with your submission that you weren’t able to fix or to give us general feedback about the problem set.