COMP 302 Winter 2011

Mid-term Examination, 14 Feb 2011

Jesse Doherty

NAME:

ID:

This examination is open book. No calculators, laptops or cell phones are allowed. You have 45 minutes. **Please write your answers on the exam paper. Good luck!**

Question 1[15 points]

Implement a function `multf:int -> (int -> int) -> int` which takes in as argument a positive integer `k` and *returns a function* which when given a function `f` it will compute the product `f(1) * f(2) * ... * f(k)`. Stage your computation in such a way that you will return a function which generates the code for `f(1) * f(2) * ... * f(k)`, i.e. the result you return should be independent of the function `multf`.

In OCaml:

```
let rec multf k f =
  if k < 1 then failwith "neg int"
  else if k = 1 then (f 1)
  else (f k) * (multf (k-1) f);;
```

fun multf (k) = (fn (f:int->int) => f(k) * multf(k-1))

multf(0) = 1

types

not staged

task problem about induction
task about options.
Datatypes Ch. 6 slide 15.

fun multf k =
  case k of 1 => f(1)
  | f => f(k) *1 multf(k-1)

f(k) * f(k-1) ... f(1)

Question 2[30 points]: Vectors

Q 2.1[15 points]
Implement a function

```
scale:'a vector * ('a -> 'b)  -> 'b vector
```

which takes in a vector $v = (x_1\ x_2\ x_3 \ldots)$ and a function f of type 'a -> 'b and scales every element x_i of the vector by applying the function f. You can assume that $f(0) = 0$.
For example:

```
scale (Vec(1, Vec(2, Vec(3 , Zeros))), fn x => x * 2)
```

will return the vector `Vec(2, Vec(4, Vec(6 , Zeros)))`

bad syntax

Scale('a vector, F: 'a -> 'b) = F('a) * scale(vector, F)
why multiplication
| scale(zeros, f) = zeros ~

In Ocaml:

```
type vector =
| Vec of int * vector
| Zeros ;;
let rec ((v: vector), (f: int → int)) =
 match v with
| Zeros → Zeros.
| Vector(x, v2) → Vec(fx, scale(v2, f);;
```

Q 2.2[15 points] Implement a function

```
comp:'a vector * 'a vector * ('a * 'a -> 'a) -> 'a vector
```

which takes in two vectors v_1 and v_2 and composes them pointwise via the function f. Given two vectors $v_1 = (x_1\ x_2\ \ldots)$ and $v_2 = (y_1\ y_2\ \ldots)$ and a function f of type 'a * 'a -> 'a, it returns vector $v = (w_1\ w_2\ \ldots)$ where the $i-th$ position $w_i = f(x_i, y_i)$. You can assume that the function f will have the property: $f(0,x) = x$ and $f(x,0) = x$.

For example:

```
comp (Vec(1, Vec(2, Vec(3,Zeros))),
      Vec(2, Vec(1, Zeros)), (fn (x,y) => (x+y)))
```

will evaluate to `Vec(3,Vec(3,Vec(3,Zeros)))`

syntax !

comp('a·veca, 'b·vecb) = (a+b)·comp(veca, vecb)

| comp(zeros, 'b·vecb) = 'b·comp(zeros, vecb) ✓

| comp('a·veca, zeros) = 'a·comp(veca, zeros) ✓

} do not need to recurse

| comp(zeros, zeros) = zeros ✓

where is the function ('a × 'a → 'a) ?

In OCaml:

(* type vector from before *)

let rec comp ((v1 : vector), (v2 : vector), f) =
match (v1, v2) with
| (Zeros, Zeros) → Zeros
~~| (Zeros, Vec (x, Zeros)) | (Vec (x, Zeros), Zeros) → Vec (f(x,0), Zeros)~~
| (Zeros, Vec (x, v)) | (Vec(x,v), Zeros) → Vec(f(x,0), comp(v, Zeros, f))
| (Vec (x1, v3), Vec(x2, v4) → Vec(f(x1, x2), comp(v3, v4, f))

fact_tr is the tail-recursive version of fact.
acc will be the accumulator holding the product of the initial value of acc and n! ~~factor~~. Therefore, if the initial value of acc = 1, then fact-tr(1, n) = 1*n!.

Question 3[35 points]

Consider the following two programs for factorial.

```
fun fact n =
if n = 0 then 1
else n * fact(n-1)
```

```
fun fact_tr (n, acc) =
if n = 0 then acc
else fact_tr(n-1, acc*n)
```

I.H

Prove that fact(n) = fact_tr(n, 1). State carefully your theorem, and your induction hypothesis.

Generalizing the induction hypothesis, which is...

IH: $\forall x$, fact(n) = fact_tr(n, x)

fact(n+1) = fact_tr(n+1, 1)

Basecase: n=1

fact(1) = 1 · fact(1-1) = 1 · fact(0) = 1 · 1 = 1 } by program

IH: ① Assume fact(n) = ... fact(n') = ...
② Assume: fact_tr(1, 1) = fact_tr(1-1, 1·1) = fact_tr(0, 1) = 1 } by program

doesn't look generalized

Step case: n+1 : assume fact(n) = fact_tr(n, 1)

fact(n+1) = (n+1)*fact(n+1-1) by program
= (n+1) fact(n) by simplification

fact_tr(n+1, acc) = fact_tr(n+1-1, acc*(n+1))
= (n+1) * fact_tr(n, acc) – by identity,
X program returns acc. thus,
fact_tr(n, acc*k)
k*fact_tr(n, acc)

(n+1) fact(n) = (n+1) fact_tr(n, acc) – by induction hypothesis?

Base case: for n = 0
fact 0 = 1   fact_tr(0, 1) = 1 ✓
with k ∈ ℕ,
Inductive step: Assume for k ≤ n that fact k = fact_tr(k, 1) is true.
fact k = fact_tr(k, 1)
×k+1 ⟺ (fact k) × (k+1) = (fact_tr(k, 1)) × (k+1)
(fact k) × k+1 = ~~fact(k+1)~~ k! × (k+1) = (k+1)! = fact(k+1)
(fact_tr(k, 1)) × (k+1) = ~~fact~~ tr 1 × k! × (k+1) = (k+1)! = fact_tr(k+1, 1)
⟹ (k+1)! = fact_tr(k+1, 1) = fact(k+1)
Therefore, if prop true for k+1, then property true for all n ∈ ℕ, QED. □   4

Question 4 [20 points]: References

Often we want to instrument programs to monitor their behaviour and performance. In this question, you are asked to implement the following function

```
mon_ref:'a  -> ((unit -> int) * (unit -> 'a) * ('a -> unit))
```

The intention is to monitor read and write accesses to a reference cell. The function `mon_ref` accepts an initial value `a` of type `'a` as input, initializes a new cell `r` with the value `a` and returns a triple:

- The first component is a function of type `unit -> int`. It returns how often we have accessed the reference cell `r`.
- The second component is a function of type `unit -> 'a`. It will read the current value in the cell `r` and return it. At the same time, it will increment the counter.
- The third component is a function `'a -> 'unit` which writes the value of type `'a` to the cell `r`. As a side effect, it will also increment the counter to keep track of how many times we have accessed the cell `r`.

Your code should count how often we access the cell from the top level, and we want to have different counters for different cells. To illustrate, consider the following example, where we monitor how often we access memory:

```
- val (cntr, c_read, c_write) = mon_ref 5;
val cntr = fn : unit -> int
val c_read = fn : unit -> int
val c_write = fn : int -> unit
```

To illustrate the behaviour of count, `c_read`, and `c_write` we give an example.

```
- cntr ();
val it = 0 : int
- c_read ();
val it = 5 : int
- cntr ();
val it = 1 : int
- c_write 7;
val it = () : unit
- cntr ();
val it = 2 : int
- c_read ();
val it = 7 : int
- c_read ();
val it = 7 : int
- c_read ();
val it = 7 : int
- cntr ();
val it = 5 : int
```

fun cntr(a:ref) = ref a ; = 1 X

①
```
type func =
 | Cntr of unit
 | Cread of unit
 | Cwrite of unit;;

let mon_ref a =
let num = ref a in
let cnt = ref 0 in
fun (f:func) →
 match f with
 | Cntr () → !cnt
 | Cread () → ((cnt := (!cnt)+1); !num)
 | Cwrite(b) → ((cnt := (!cnt)+1); num := b; !num);;
```

Pb: Works perfectly, but takes a special input (func), does not return the triple of function, and can't have ≠ types for the output of the function

②
```
let mon_ref a =
let num = ref a in
let cnt = ref 0 in
let fs = ((let cntr() = !cnt in cntr),
(let c_read() = ((cnt := (!cnt)+1); !num)),
(let c_write b = ((cnt := (!cnt)+1); num := b));;
```

Pb: We get exactly what we want, but can't do shit!

Question 2[30 points]: Vectors

Q 2.1[15 points]

Implement a function

`scale:'a vector * ('a -> 'b) -> 'b vector`

which takes in a vector $v = (x_1\ x_2\ x_3 \ldots)$ and a function f of type `'a -> 'b` and scales every element x_i of the vector by applying the function f. You can assume that $f(0) = 0$.

For example:

`scale (Vec(1, Vec(2, Vec(3 , Zeros))), fn x => x * 2)`

will return the vector `Vec(2, Vec(4, Vec(6 , Zeros)))`

bad syntax

Scale(a::vector, f:(a->b)) = f(a)::Scale(vector, f)

|Scale(Zeros, f) = Zeros

why multiplication

0.