Comp 302 Final

Concepts

let curry f = (**fun** x y −> f (x, y))
let curry2 f x y = f (x, y)
let curry3 = **fun** f −> **fun** x −> **fun** y −> f (x, y)
let uncurry f = (**fun** (x, y) −> f x y)

(* Functions are right associative *)
(* Functions are not evaluated until they need to be *)
let test a b = a * a + b
test 3 = **fun** y −> 3 * 3 + y (* Not 9 + y *)

Syntax

Do not forget about 'rec', 'let ... in', brackets, constructors or tuples

match x with
| a −> (* return *)
| b −> (* Nested matching *)
  begin match ... **with**
   | ... −>
  end
| _ −> (* wildcard return *)

let name arg1 arg2 =
  let inner' arg1' arg2' = out' **in**
  inner' arg1 arg2

exception Failure **of** string
raise (Failure "what_a_terrible_failure")

let x = 2 **and** y = 4 (* Initializes both *)

(* ('a * 'b −> 'c) −> 'a −> 'b −> 'c = <fun> *)
let cur = **fun** f −> **fun** x −> **fun** y −> f (x,y)

(* 'a list list −> 'a list = <fun> *)
let first lst = **match** lst **with**
| [] −> [] | x::xs −> x

(* val is_zero : int −> string = <fun> *)
let is_zero = **function** | 0 −> "zero" | _ −> "not_zero"

(* Variable bindings are overshadowed;
bindings are valid in their respective scopes *)
let m = 2;; let m = m * m **in** m (* is 4 *);;
m (* is 2 *);; let f () = m;; let m = 3;; f () (* is 2 *);;

List Ops

elem :: list   list1 @ list2
val length : 'a list −> int
val find_opt : ('a −> bool) −> 'a list −> 'a option
val filter : ('a −> bool) −> 'a list −> 'a list
val map : ('a −> 'b) −> 'a list −> 'b list
val fold_right : ('a −> 'b −> 'b) −> 'a list −> 'b −> 'b
val for_all : ('a −> bool) −> 'a list −> bool
val exists : ('a −> bool) −> 'a list −> bool
val rev : 'a list −> 'a list
val init : int −> (int −> 'a) −> 'a list (* by index *)

Types & Option

(* Base types: bool, int, char, float, 'a list, option *)
(* 'x denotes a polymorphic type (Java Generics) *)
type 'a option = None | Some **of** 'a
(* Constructors can be used to match within types
match cases are sufficient once all constructors are matched *)
type rational = Integer **of** int
  | Fraction **of** rational * rational
type 'param int_pair = int * 'param
let x = (3, 3.14) (* val x : int * float = 3, 3.14 *)
(* Valid specified type *)
let (x : int_pair) = (3, 3.14) (* val x : int_pair = 3, 3.14 *)

Higher Order Functions

(* sum : (int −> int) −> int * int −> int *)
let **rec** sum f (a, b) =
  if a > b **then** 0
  else f a + sum f (a + 1, b)
(* sumCubes : int * int −> int = <fun> *)
let sumCubes (a, b) = sum (**fun** x −> x * x * x) (a, b)

Induction

- Mathematical, Structural
- Can generalize theory when proving
  Eg rev (x::t) = rev' (x::t) [] ⇒ rev l @ acc = rev' l acc

$e \Downarrow v$ multi step evaluation from e to v
$e \Rightarrow e'$ single step evaluation from e to e'
$e \Rightarrow^* e'$ multiple small step evaluations from e to e'

State theory and IH; do base case

let **rec** even_parity = **function**
| [] −> **false**
| **true**::xs −> not (even_parity xs)
| **false**::xs −> even_parity xs

let even_parity_tr l = let **rec** parity p = **function**
| [] −> p | p'::xs −> parity (p<>p') xs **in**
  parity **false** l

(* IH: For all l, even_parity l = even_parity_tr l *)
(* Case for true: *)
even_parity_tr **true**::xs
= parity **false true**::xs   (* Def of even_parity_tr *)
= parity (**false** <> **true**) xs (* Def of parity *)
= parity **true** xs   (* Def of <> *)
= not (parity **false** xs)   (* Prove? *)
= not (even_parity_tr xs) (* Def of even_parity_tr *)
= not (even_parity xs)   (* IH *)
= even_parity **true**::xs   (* Def of even_parity *)

References

mutable type	type t = { mutable i : int }	
init	let x = ref 0	let y = i = 5
retrieval	let val1 = !x	let val2 = y.i
assignment	x := 9	y.i <−1

```
module type STACK =
sig
    type stack
    type t
    val empty : unit −> stack
    val push : t −> stack −> stack
    val size : stack −> int
    val pop : stack −> stack option
    val peek : stack −> t option
end

module IntStack : (STACK with type t = int) =
struct
    type stack = int list
    type t = int
    let empty () = []
    let push i s = i :: s
    let size = List.length
    let pop = function | [] −> None | _ :: t −> Some t
    let peek = function | [] −> None | h :: _ −> Some h
end
```

(* Susp *)

```
type 'a susp = Susp of (unit −> 'a)
type 'a str = {hd: 'a; tl : ('a str) susp}
let delay f = Susp f        (* (unit −> 'a) −> 'a susp *)
let force (Susp f) = f()    (* 'a susp −> 'a *)
```

(* ('a −> 'b −> 'c) −> 'a str −> 'b str −> 'c str *)
```
let rec zip f s1 s2 = {hd = f s1.hd s2.hd ;
    tl = delay (fun () −> zip f (force s1.tl) (force s2.tl)) }
```

(* Sieve of Eratosthenes *)

```
let rec filter_str  (p : 'a −> bool) (s : 'a str) =
    let h, t = find_hd p s in {hd = h;
    tl = delay (fun () −> filter_str  p (force t))}
and find_hd p s = if p s.hd then (s.hd, s.tl)
    else find_hd p (force s.tl)

let no_divider m n = not (n mod m = 0)
let rec sieve s = { hd = s.hd;
    tl = delay (fun () −>
        sieve ( filter_str   (no_divider s.hd) (force s.tl) ))}
```

(* val double : ('a −> 'a) −> 'a −> 'a = <fun> *)
```
let double = fun f −> fun x −> f(f(x))
```

Misc

An effect is an action resulting from evaluation of an expression other than returning a value.

(* Coin *)

exception BackTrack

(* val change : int list −> int −> int list = <fun> *)
```
let rec change coins amt = if amt = 0 then []
    else (match coins with
        | [] −> raise BackTrack
        | coin :: cs −>
            if coin > amt then change cs amt
            else try coin :: (change coins (amt − coin))
                with BackTrack −> change cs amt)
```

(* val change : int list −> int −>
    (int list −> 'a) −> (unit −> 'a) −> 'a = <fun> *)
```
let rec change coins amt success failure =
    if amt = 0 then success []
    else match coins with
        | [] −> failure ()
        | coin :: cs −>
            if coin > amt then change cs amt success failure
            else change coins (amt − coin)
                (fun list −> success (coin :: list ))
                (fun () −> change cs amt success failure)
```

Syntax & Semantics

- Definition
  $FV((e_1, e_2)) = FV(e_1) \cup FV(e_2)$
- Substitution
  $[e'/x'](\text{let pair } (x, y) = e_1 \text{ in } e_2 \text{ end}) = \text{let pair } (x, y) = [e'/x']e_1 \text{ in } [e'/x']e_2 \text{ end}$
  Provided $x' \neq x$ && $x' \neq y$ && $(x, y \neq FV(e'))$
- Type
  Types $t ::= \text{int} \mid \text{bool} \mid T_1 \to T_2 \mid T_1 \times T_2 \mid \alpha$
  $$\frac{\Gamma \vdash (x, y) : T \times S \quad \Gamma \cup \{(x : T), (y : S)\} \vdash e_2 : U}{\Gamma \vdash \text{let pair } (x, y) = e_1 \text{ in } e_2 : U}$$
  T-LET
  Provided $x' \neq x, x' \neq y, \& x, y \notin FV(e')$

  My Solution (they don't account for e_1)
  $$\frac{\Gamma \vdash e_1 : T_1 \times T_2 \quad \Gamma, x : T_1, y : T_2, \vdash e_2 : T}{\Gamma \vdash \text{ let } (x, y) = e_1 \text{ in } e_2 \text{ end} : T}$$
  T-LET-MATCH
- Operation
  $$\frac{e_1 \Downarrow (v_1, v_2) \quad [v_1/x][v_2/y]e_3 \Downarrow v}{\text{let pair } (x, y) = e_1 \text{ in } e_2 \Downarrow v_3} \text{ B-LET}$$
- References
  $$\frac{\Gamma \vdash e_1 : T \text{ ref} \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : \text{ unit}} \qquad \frac{\Gamma \vdash e : T \text{ ref}}{\Gamma \vdash !e : T}$$
  $$\frac{\Gamma \vdash e : T}{\Gamma \vdash \text{ref } e : T \text{ ref}} \qquad \frac{}{\Gamma \vdash () : \text{ unit}}$$
- Preservation: If $e \Downarrow v$ and $e : T$ then $v : T$

$\Gamma \vdash e \Rightarrow T/C$: Infer type T for expression e in the typing environment Γ module the constraints C $\dfrac{x : T \in \Gamma}{\Gamma \vdash x \Rightarrow T/\emptyset}$ B-VAR

$$\frac{\Gamma \vdash \Rightarrow T/C \quad \Gamma \vdash e_1 \Rightarrow T_1/C_1 \quad \Gamma \vdash e_2 \Rightarrow T_2/C_2}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow T_1/C \cup C_1 \cup C_2 \cup \{T = \text{bool}, T_1 = T_2\}}$$
B-IF