CS 320: Mock Mid-Term

Total: 100 pts

CS 320 Course Staff

1 Weird Programming Languages [50 pts]

In this section, we will design slightly different programming languages than (the subset of) OCaml that we know and love. As usual, this would mean defining the formal aspects of these languages: syntax, type system, and semantics. For your benefit, we will be defining some parts of the formalism and relying on you to fill in the missing details.

The syntax of the language looks exactly like we have in OCaml. We won't write all the expressions here, just the ones relevant for the following problems.

$$\langle expr \rangle ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ \langle expr \rangle\ \mathsf{then}\ \langle expr \rangle\ \mathsf{else}\ \langle expr \rangle \mid n \mid \langle expr \rangle + \langle expr \rangle \mid \ldots$$

In this syntax, true and false represent the standard boolean values, and n stands for integer constants (e.g., 0, 1, -1, etc.). The if and addition expressions are also standard.

1.1 Integers Inside If Expressions

For the expression if e then e_1 else e_2, we usually assume that e has type bool. Hence, the typing rule is written as

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}\ \textsc{If}$$

and the semantics rules are written as

$$\frac{e \Downarrow \mathsf{true} \qquad e_1 \Downarrow v_1}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v_1}\ \textsc{If-Then} \qquad\qquad \frac{e \Downarrow \mathsf{false} \qquad e_2 \Downarrow v_2}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v_2}\ \textsc{If-Else}$$

Suppose, instead, we allowed expression e to be an integer also with the following behavior: If e is positive (i.e., $e \geq 0$), then the expression goes into evaluating the then branch. Otherwise, if e is negative, then the expression goes into evaluating the else branch.

Problem 1 (5 pts) *Write the corresponding typing rule(s) for this new* if *expression.*

Solution.

$$\frac{\Gamma \vdash e : \mathsf{int} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}\ \textsc{If}$$

Problem 2 (5 pts) *Write the corresponding semantics rule(s) for this new* if *expression.*

Solution.

$$\frac{e \Downarrow v \qquad v \geq 0 \qquad e_1 \Downarrow v_1}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v_1}\ \textsc{If-Then} \qquad\qquad \frac{e \Downarrow v \qquad v < 0 \qquad e_2 \Downarrow v_2}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v_2}\ \textsc{If-Else}$$

Problem 3 (10 pts) *Consider the following expression:*

$$\mathsf{if}\ 1 + 2\ \mathsf{then}\ 3\ \mathsf{else}\ 4$$

Is this expression valid (i.e., well-typed)? If yes, please show a valid typing derivation. Otherwise, give an intuitive explantation of why it's not well-typed.

Solution. Yes, this is a valid expression.

$$\dfrac{\dfrac{\dfrac{}{\cdot \vdash 1 : \mathsf{int}} \quad \dfrac{}{\cdot \vdash 2 : \mathsf{int}}}{\cdot \vdash 1 + 2 : \mathsf{int}}\ \textsc{IntAdd} \quad \dfrac{}{\cdot \vdash 3 : \mathsf{int}} \quad \dfrac{}{\cdot \vdash 4 : \mathsf{int}}}{\cdot \vdash \mathsf{if}\ 1 + 2\ \mathsf{then}\ 3\ \mathsf{else}\ 4 : \mathsf{int}}\ \textsc{If}$$

Problem 4 (10 pts) *If it's valid, what is the value of the expression above from Problem 3? Show a semantic derivation. If the expression is invalid, can you fix it so that it's valid and then do a semantic derivation to compute its value?*

Solution.

$$\dfrac{\dfrac{1 \Downarrow 1 \quad 2 \Downarrow 2}{1 + 2 \Downarrow 3}\ \textsc{IntEval} \quad 3 \geq 0 \quad 3 \Downarrow 3}{\mathsf{if}\ 1 + 2\ \mathsf{then}\ 3\ \mathsf{else}\ 4 \Downarrow 3}\ \textsc{If-Then}$$

1.2 Adding Integers and Floats

Suppose we allowed adding integers and floats in our language using the + operator such that if either of the summands is a float, then the result is a float.

Problem 5 (10 pts) *Write all possible typing rule(s) for this new addition expression.*

Solution.

$$\dfrac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}\ \textsc{AddIntInt} \qquad \dfrac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{float}}{\Gamma \vdash e_1 + e_2 : \mathsf{float}}\ \textsc{AddIntFloat}$$

$$\dfrac{\Gamma \vdash e_1 : \mathsf{float} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{float}}\ \textsc{AddFloatInt} \qquad \dfrac{\Gamma \vdash e_1 : \mathsf{float} \quad \Gamma \vdash e_2 : \mathsf{float}}{\Gamma \vdash e_1 + e_2 : \mathsf{float}}\ \textsc{AddFloatFloat}$$

Problem 6 (10 pts) *With the* if *expression updated according to Section 1.1, consider the following expression:*

$$\mathsf{if}\ 1 + 2.0\ \mathsf{then}\ 3\ \mathsf{else}\ 4$$

Is this expression valid (i.e., well-typed)? If yes, please show a valid typing derivation. Otherwise, give an intuitive explantation of why it's not well-typed.

Solution. No, this is not a well-typed expression. The expression $1 + 2.0$ has type float but the if e then e_1 else e_2 expression expects e to have type bool or int but not float.

2 Polymorphic Higher-Order Programming [20 pts]

Problem 7 (20 pts) *Below, we will provide a list of function types. Either define a function of this type, or briefly explain why it is impossible to do so. You are welcome to use the standard library functions that we have discussed in the course.*

1. $\mathsf{int} \to \text{`}a\ \mathsf{list}$
2. $\text{`}a \to \mathsf{int}$

3. $('a \to 'b \to 'a) \to 'a\ \texttt{list} \to 'b\ \texttt{list}$

4. $'a \to 'a\ \texttt{list}$

5. $'a\ \texttt{list} \to 'a$

Solution.

1. `let f _ = []`

2. `let f _ = 0`

3. `let f _ _ = []`

4. `let f x = [x]`

5. Not possible since input list can be empty

3 Good Old OCaml Programming [30 pts]

No midterm is complete without some OCaml programming. Implement the following functions. Again, you are welcome to use standard library functions unless explicitly stated. You are also welcome to define helper functions.

Problem 8 (10 pts) *Define a function called* `merge_sort` *that sorts an input list*

$$\texttt{val merge\_sort} : 'a\ \texttt{list} \to 'a\ \texttt{list}$$

using the standard merge sort algorithm. You cannot (obviously) use the `List.sort` *and* `List.merge` *functions.*

Solution.

```
let rec split l =
  match l with
  | [] -> ([], [])
  | [h] -> ([h], [])
  | h1::h2::t ->
      let (t1, t2) = split t in
      (h1::t1, h2::t2)

let rec merge l1 l2 =
  match l1, l2 with
  | [], [] -> []
  | [], _ -> l2
  | _, [] -> l1
  | h1::t1, h2::t2 ->
      if h1 < h2 then h1::(merge t1 (h2::t2))
      else h2::(merge (h1::t1) t2)

let rec merge_sort l =
  match l with
  | [] -> []
  | [x] -> [x]
  | _ ->
      let (l1, l2) = split l in
      merge (merge_sort l1) (merge_sort l2)
```

Problem 9 (10 pts) *Define a function called* `pack` *which packs together consecutive elements of a list that are equal into a sublist.*

$$\mathsf{val\ pack} : \text{`}a\ \mathsf{list} \to \text{`}a\ \mathsf{list\ list}$$

For instance, with the input $[1;1;2;3;3;2;4;4;4]$*, the output should be* $[[1;1];[2];[3;3];[2];[4;4;4]]$*. Note that the second occurrence of 2 is not packed with the first occurrence of 2 since only* ***consecutive*** *elements that are equal are packed together.*

Solution.

```
let rec pack_helper l num acc curr_list =
  match l with
  | [] -> List.rev ((num::curr_list)::acc)
  | h::t ->
    if h = num
    then pack_helper t num acc (h::curr_list)
    else pack_helper t h ((num::curr_list)::acc) []

let pack l =
  match l with
  | [] -> []
  | h::t -> pack_helper t h [] []
```

Problem 10 (10 pts) *Define a function called* `factorize` *which computes the prime factors of the input and produces a list of tuples containing the prime factor and the corresponding exponent.*

$$\mathsf{val\ factorize} : \mathsf{int} \to \mathsf{int} * \mathsf{int\ list}$$

For input n*, if the number* n *can be written as* $p_1^{e_1} \cdot p_2^{e_2} \cdots p_n^{e_n}$*, then the output should be the list* $[(p_1, e_1); (p_2, e_2), \ldots, (p_n, e_n)]$*. Example:* `factorize` $360 = [(2,3);(3,2);(5,1)]$*. The prime numbers in the list must be in increasing order. For input* 1*, just return the empty list. Your implementation only needs to work for positive numbers.*

Solution.

```
let rec is_prime_helper n m limit =
  if m > limit
  then true
  else if n mod m = 0
  then false
  else is_prime_helper n (m+1) limit

let rec is_prime n =
  let limit = int_of_float (sqrt (float_of_int n)) in
  is_prime_helper n 2 limit

let rec compute_exp n p t =
  if n mod p = 0
  then compute_exp (n / p) p (t+1)
  else (n, t)

let rec factorize_helper n p acc limit =
  if p > limit
  then List.rev acc
  else if is_prime p
  then
    let (q, e) = compute_exp n p 0 in
    if e > 0
    then factorize_helper q (p+1) ((p, e)::acc) limit
```

```
    else factorize_helper n (p+1) acc limit
  else
    factorize_helper n (p+1) acc limit

let factorize n =
  factorize_helper n 2 [] n
```