Practice Midterm Examination

CAS CS 320: Principles of Programming Languages

February 20, 2025

Name:

BUID:

- ▷ You will have approximately 75 minutes to complete this exam. Make sure to read every question, some are easier than others.
- ▷ Do not remove any pages from the exam.
- ▷ Make very clear what your final solution for each problem is (e.g., by surrounding it in a box). We reserve the right to mark off points if we cannot tell what your final solution is.
- ▷ You must show your work on all problems unless otherwise specified. A solution without work will be considered incorrect (and will be investigated for potential academic dishonesty).
- ▷ Unless stated otherwise, you should only need the rules provided **in that problem** for your derivations.
- ▷ We will not look at any work on the pages marked "*This page is intentionally left blank.*" You should use these pages for scratch work.

1 Repeats

Without using any functions from the standard library, implement the function

```
val repeats :  ('a * int) list → 'a option list
```

so that `repeats l` is the result of replacing each tuple `(x, n)` with `n` copies of `Some x` in the case that `n` is nonnegative and `-n` copies of `None` otherwise. **Your implementation must be tail recursive.**

```
let _ = assert (repeats [(true, 3)]
                = [Some true; Some true; Some true])
let _ = assert (repeats [(1, 2); (2, 0), (3, -3)]
                = [Some 1; Some 1; None; None; None])
```

2 Merge Sort

Consider the following partial implementation of merge sort using a specialized ADT called `merge_list`.

```
let rec append (x : 'a) (l : 'a merge_list) : 'a merge_list = match l with
  | Nil -> Single x
  | Single y -> Merge {left=Single x;right=Single y}
  | Merge {left;right} -> Merge {left=right;right=append x left}

let rec of_list l = match l with
  | [] -> Nil
  | x :: xs -> append x (of_list xs)

let rec merge l r = assert false

let rec merge_sort (l : 'a list) : 'a list =
  let rec go l = match l with
    | Nil -> []
    | Single x -> [x]
    | Merge ls -> merge (go ls.left) (go ls.right)
  in go (of_list l)
```

A. Based on the above code, give the definition of the `merge_list` type.

B. Implement the function

```
val merge :  'a list -> 'a list -> 'a list
```

so that `merge l r` is the sorted list with the same elements as `l @ r`, **assuming `l` and `r` are already sorted.** You may not use any functions from the standard library except for comparison functions like `(<)`.

(Problem Continued)

3 Typing Derivations

A. Write down an expression of type `('a * 'b) → ('b * 'a)`.

B. Let e denote the expression you wrote down in the previous part. Write a derivation of the judgment

$$\cdot \vdash e : (\tau_1 \texttt{ * } \tau_2) \texttt{ → } (\tau_2 \texttt{ * } \tau_1)$$

where your derivation should be written in terms of τ_1 and τ_2.[1]

[1] On the actual exam we will make the rules available.

4 Alternating Paths

Consider the following ADT for a binary tree.

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

We can think of a path in a binary tree from the root of the tree to a leaf as a sequence of "lefts" and "rights", i.e., whether the path goes down a left subtree or a right subtree. The *alternation number* of a path is the number of times the path went "left" after going "right" or vice versa. The alternation number of a tree is the maximum alternation number over all paths from the root to a leaf in the tree. Implement the function

```
val alt_num :  'a tree -> int
```

so that `alt_num t` is the alternation number of the tree `t`. You may not use any function in the standard library except `max`. *Hint:* Write a helper function that returns *two* values instead of one.

(Problem Continued)

5 Options, Formally

We've seen option types in OCaml, but we did not include the typing rules in our `320Caml` specification.

A. In analogy with lists, provide the typing rules for option types. Recall that options are defined by the following ADT

```
type 'a option =
  | None
  | Some of 'a
```

B. Give the typing rule for shallow pattern matching on options. That is, write down the rules for determining how to type an evaluate an expression of the following form:

```
match o with | None -> none_case | Some n -> some_case
```

6 Semantic Derivation

Give a derivation of the following semantic judgment.

$$\texttt{let x = 2 in let z = x + x in (x * z, z)} \Downarrow (8, 4)$$