Final Examination

CAS CS 320: Principles of Programming Languages

May 6, 2025

Name:

BUID:

- ▷ You will have approximately 120 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.
- ▷ Unless stated otherwise, you should only need the rules provided in the problem statement for derivations. Side conditions of rules appear in shaded boxes.
- ▷ 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.

Problem #	Points
1	12
2	8
3	12
4	5
5	13

This page is intentionally left blank.

1 Association Lists

In our mini-projects, we implemented typing contexts and dynamic environments using maps. We could have alternatively used association lists, i.e., lists of key-value pairs. **For all parts of this problem, you may *not* use anything from the OCaml standard library except for constructors like `[]`, `(::)`, `None`, and `Some`.** In particular, you may not use the function `(@)`.

We say that an association list `l` satisfies the *unique keys property* if no key appears more than once in `l`. For example, `[(1, 2); (2, 3); (3, 2)]` satisfies the unique keys property, but `[(1, 2); (1, 3)]` does not, and neither does `[(1, 2); (2, 3); (1, 2)]`.

A. (4 points) Implement the function `find_opt` so that `find_opt k l` is `Some v` if `(k, v)` is an element of `l`, and is `None` otherwise. You may assume that `l` satisfies the unique keys property.

```
val find_opt : 'k -> ('k * 'v) list -> 'v option

(* let _ = assert (find_opt "1" [] = None) *)
(* let _ = assert (find_opt "1" [("2", 2); ("1", 1)] = Some 1) *)
(* let _ = assert (find_opt "1" [("2", 2); ("one", 1)] = None) *)
```

B. (3 points) Implement the function `rev_append` so that `rev_append l r` is equivalent to `(List.rev l) @ r`. In other words, `rev_append l r` is the result of appending the reverse of `l` to `r`. **Your implementation must be tail-recursive.**

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

(* let _ = assert (rev_append [1;2;3] [4;5;6] = [3;2;1;4;5;6]) *)
```

C. (5 points) Implement the function `add` so that `add k v l` is the association list `l` but with the mapping `(k, v)`. You may assume that `l` satisfies the unique keys property. Your implementation should additionally satisfy the following properties.

(a) If `k` appears as a key in `l`, then its value is replaced with `v` in `add k v l`.

(b) If `k` does not appear as a key in `l`, then the last element of `add k v l` is `(k, v)`.

(c) The order of the elements in `l` are preserved.

(d) The list `add k v l` satisfies the unique keys property.

(e) **Your implementation must be tail recursive.**

Use may use the function `rev_append` from the previous part without reimplementing it, even if you didn't complete the previous part.

```
val add : 'k -> 'v -> ('k * 'v) list -> ('k * 'v) list

(* let _ = assert (add 1 true [] = [(1, true)]) *)
(* let _ = assert (add 1 true [(2, false)] = [(1, true); (2, false)]) *)
(* let _ = assert (add 1 true [(2, false); (1; false)] = [(2, false); (1, true)]) *)
(* let _ = assert (add 1 1 [(2, 0); (1; 0); (3, 1)] = [(2, 0); (1, 1); (3, 1)]) *)
(* let _ = assert (add 1 1 [(2, 0); (3, 1)] = [(1, 1); (2, 0); (3, 1)]) *)
```

Problem Continued

2 Grammatical Ambiguity

Let $\mathcal{G}$ denote the following grammar. The production rule for <c> has the terminal symbol '|' which we put in quotes to distinguish it from the BNF alternative symbol. The quotes should not appear in any derivation or parse tree for this problem.

<e> ::= match <e> with | match <e> with <cs> | x | true | false | (<e>)
<cs> ::= <c> | <c> <cs>
<c> ::= '|' <p> → <e>
<p> ::= x | true | false

A. (4 points) Demonstrate that $\mathcal{G}$ recognizes the following sentence by writing a derivation of it *or* drawing a parse tree for it (*Remember.* In a derivation, you can only expand one nonterminal symbol of a sentential form at a time).

match x with | true → false | false → true

```
<e> ::= match <e> with  | match <e> with <cs> | x | true | false | (<e>)
<cs> ::= <c> | <c> <cs>
<c> ::= '|' <p> → <e>
<p> ::= x | true | false
```

B. (4 points) Demonstrate that $\mathcal{G}$ (reproduced above) is ambiguous by determining a *shortest* sentence (in number of terminal symbols) with two distinct parse trees in $\mathcal{G}$. Additionally, write a one-sentence English explanation for what causes the ambiguity. You are not required to write any derivations or draw any parse trees. (*Note.* Even in OCaml, match statements can be empty.)

3 Let-Polymorphism

A. (6 points) Let α, β, γ, δ and η be type variables. Write a derivation of the following judgment in **compact form**. You do not need to label your inferences with rule names but it can be helpful for partial credit.

$$\{\texttt{x} : \alpha \ , \ \texttt{k} : \beta \to \gamma \to \beta\} \ \vdash \ \texttt{let y = k x x in k} : \beta \to \gamma \to \beta \ \dashv \ \delta \doteq \alpha \to \eta \ , \ \beta \to \gamma \to \beta \doteq \alpha \to \delta$$

You may use the symbol 'Γ' for the context $\{\texttt{x} : \alpha \ , \ \texttt{k} : \beta \to \gamma \to \beta\}$, but otherwise do not use any shorthands in your derivation. The rules required to complete this derivation are given here for reference.

$$\frac{(x : \sigma) \in \Gamma \qquad \sigma \text{ is a monotype}}{\Gamma \vdash x : \sigma \dashv \varnothing} \text{ (var-m)} \qquad \frac{\alpha \text{ is fresh} \qquad \Gamma, x : \alpha \vdash e : \tau \dashv \mathcal{C}}{\Gamma \vdash \texttt{fun}\ x \to e : \alpha \to \tau \dashv \mathcal{C}} \text{ (fun)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \alpha \text{ is fresh}}{\Gamma \vdash e_1\ e_2 : \alpha \dashv \tau_1 \doteq \tau_2 \to \alpha, C_1, \mathcal{C}_2} \text{ (app)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \dashv \mathcal{C}_1, \mathcal{C}_2} \text{ (let)}$$

Problem continued

B. (6 points) Let e denote the following expression:

```
fun x ->
  let k = fun x -> fun y -> x in
  let y = k x x in
  k
```

The following judgment is derivable according to the rules given in the previous part.

$$\cdot \vdash e : \alpha \to \beta \to \gamma \to \beta \dashv \delta \doteq \alpha \to \eta \ , \ \beta \to \gamma \to \beta \doteq \alpha \to \delta$$

From this judgment, determine the principle type of e. You must show your work to recieve full credit. *Hint.* The principle type of this expression according to the given rules is *not* the same as the type of this expression in OCaml. This is because OCaml implements let-polymorphism whereas the given rules do not.

Problem continued

4 Closures

(5 points) Determine what the following expression evaluates to in the empty environment. You are not required to write a derivation.

```
let b = 1 in
let k = fun x -> fun y -> x in
let a = fun f -> fun x -> f x in
let b = 2 in
let f = a k b in
let b = 3 in
f
```

The rules required to reason about the value of the above expression are given here for reference.

$$\frac{n \text{ is an integer}}{\langle\ \mathcal{E}\ ,\ n\ \rangle\ \Downarrow\ n}\ \text{(int-eval)} \qquad \frac{x \text{ is a variable}}{\langle\ \mathcal{E}\ ,\ x\ \rangle\ \Downarrow\ \mathcal{E}(x)}\ \text{(var-eval)}$$

$$\frac{}{\langle\ \mathcal{E}\ ,\ \texttt{fun}\ x\ \texttt{->}\ e\ \rangle\ \Downarrow\ (\!|\ \mathcal{E}\ ,\ \texttt{fun}\ x\ \texttt{->}\ e\ |\!)}\ \text{(fun-eval)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle\ \Downarrow\ (\!|\ \mathcal{E}'\ ,\ \texttt{fun}\ x\ \texttt{->}\ e\ |\!) \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle\ \Downarrow\ v_2 \qquad \langle\ \mathcal{E}'[x \mapsto v_2]\ ,\ e\ \rangle\ \Downarrow\ v}{\langle\ \mathcal{E}\ ,\ e_1\ e_2\ \rangle\ \Downarrow\ v}\ \text{(app-eval)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle\ \Downarrow\ v_1 \qquad \langle\ \mathcal{E}[x \mapsto v_1]\ ,\ e_2\ \rangle\ \Downarrow\ v_2}{\langle\ \mathcal{E}\ ,\ \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \rangle\ \Downarrow\ v_2}\ \text{(let-eval)}$$

This page is intentionally left blank.

5 References

Functional programming is great, but sometimes we want state. To this end, OCaml has *references*, which behave like pointers to locations in memory.

A. (7 points) Consider the following typing rules.

$$\frac{n \text{ is an integer}}{\Gamma \vdash n : \texttt{int}}\ (\text{int}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}\ (\text{add})$$

$$\frac{(x : \tau) \text{ appears in } \Gamma}{\Gamma \vdash x : \tau}\ (\text{var}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau_2}\ (\text{let})$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{ref } e : \tau \texttt{ ref}}\ (\text{ref}) \qquad \frac{\Gamma \vdash e : \tau \texttt{ ref}}{\Gamma \vdash \texttt{!}e : \tau}\ (\text{deref}) \qquad \frac{\Gamma \vdash e_1 : \tau \texttt{ ref} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \texttt{ := } e_2 : \texttt{unit}}\ (\text{assign})$$

Write a derivation of the following typing judgment. Your derivation should be in **standard (not compact) form**, and all inferences should be labeled with rule names. You may use the symbol 'Γ' for the context {`y : int ref, x : int ref ref`}, but otherwise do not use any shorthands in your derivation.

$$\{\texttt{y} : \texttt{int ref}\} \vdash \texttt{let x = ref y in y := !(!x) + 1} : \texttt{unit}$$

B. (6 points) Where references get interesting is in the semantics. Because references are like pointers to locations in memory, our semantics needs a notion of memory.

We take a value to be a unit ($\bullet$), an integer (e.g., 3, -23) or a *location* (usually denoted ℓ). A configuration consists of an environment $\mathcal{E}$ which binds variables to values, a store $\mathcal{M}$ which binds *locations* to values, and an expression e. A configuration is denoted as

$$\langle\ \mathcal{E}\ ,\ \mathcal{M}\ ,\ e\ \rangle$$

An evaluation judgment is of the form

$$\langle\ \mathcal{E}\ ,\ \mathcal{M}_1\ ,\ e\ \rangle\ \Downarrow\ \langle\ \mathcal{M}_2\ ,\ v\ \rangle$$

where the right-hand side is a value *together with a store*. This is necessary because the store may change as a result of evaluating the expression.

The semantic rules for references are given on the following page. Notice how the last two rules use the store to determine or update the value of a reference.

The problem continues on the following page.

$$\frac{n \text{ is an integer}}{\langle\ \mathcal{E}\ ,\ \mathcal{M}\ ,\ n\ \rangle\ \Downarrow\ \langle\ \mathcal{M}\ ,\ n\ \rangle}\ \text{(int)} \qquad \frac{x \text{ is a variable}}{\langle\ \mathcal{E}\ ,\ \mathcal{M}\ ,\ x\ \rangle\ \Downarrow\ \langle\ \mathcal{M}\ ,\ \mathcal{E}(x)\ \rangle}\ \text{(var)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ \mathcal{M}_1\ ,\ e_1\ \rangle\ \Downarrow\ \langle\ \mathcal{M}_2\ ,\ m\ \rangle \qquad \langle\ \mathcal{E}\ ,\ \mathcal{M}_2\ ,\ e_2\ \rangle\ \Downarrow\ \langle\ \mathcal{M}_3\ ,\ n\ \rangle}{\langle\ \mathcal{E}\ ,\ \mathcal{M}_1\ ,\ e_1\ \texttt{+}\ e_2\ \rangle\ \Downarrow\ \langle\ \mathcal{M}_3\ ,\ m+n\ \rangle}\ \text{(add)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ \mathcal{M}_1\ ,\ e\ \rangle\ \Downarrow\ \langle\ \mathcal{M}_2\ ,\ \ell\ \rangle}{\langle\ \mathcal{E}\ ,\ \mathcal{M}_1\ ,\ \texttt{!}e\ \rangle\ \Downarrow\ \langle\ \mathcal{M}_2\ ,\ \mathcal{M}_2(\ell)\ \rangle}\ \text{(deref)}$$

$$\frac{\langle\ \mathcal{E}\ ,\ \mathcal{M}_1\ ,\ e_1\ \rangle\ \Downarrow\ \langle\ \mathcal{M}_2\ ,\ \ell\ \rangle \qquad \langle\ \mathcal{E}\ ,\ \mathcal{M}_2\ ,\ e_2\ \rangle\ \Downarrow\ \langle\ \mathcal{M}_3\ ,\ v\ \rangle}{\langle\ \mathcal{E}\ ,\ \mathcal{M}_1\ ,\ e_1\ \texttt{:=}\ e_2\ \rangle\ \Downarrow\ \langle\ \mathcal{M}_3[\ell \mapsto v]\ ,\ \bullet\ \rangle}\ \text{(assign)}$$

Write a derivation of the following semantic judgment. Your derivation should be in **standard (not compact) form**, and all inferences should be labeled with rule names. You may use the symbol '$\mathcal{E}$' for the environment $\{\texttt{x} \mapsto \ell_1, \texttt{y} \mapsto \ell_2\}$ and the symbol '$\mathcal{M}$' for the store $\{\ell_2 \mapsto 1, \ell_2 \mapsto 2\}$, but otherwise do not use any shorthands in your derivation.

$$\langle\ \{\texttt{x} \mapsto \ell_1, \texttt{y} \mapsto \ell_2\}\ ,\ \{\ell_1 \mapsto 1, \ell_2 \mapsto 2\}\ ,\ \texttt{x := !y + 1}\ \rangle\ \Downarrow\ \langle\ \{\ell_1 \mapsto 3, \ell_2 \mapsto 2\}\ ,\ \bullet\ \rangle$$

Problem continued