Practice Midterm Exam

CAS CS 320: Principles of Programming Languages

Name: Nathan Mull

BUID: 12345678

- ▹ 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.

This page is intentionally left blank.

1 Bad Inference Rules

Recall the typing rule for functions:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e : \tau_1\ \texttt{->}\ \tau_2}\ \textsc{fun}$$

Suppose we made a mistake when writing down this typing rule, and forgot to require the type of x in the context of the premise to be the same as the argument type of `fun` x `->` e in the conclusion (the difference in the rule is boxed).

$$\frac{\Gamma, \boxed{x : \tau_1'} \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e : \tau_1 \to \tau_2}\ \textsc{funWrong}$$

One important property of a programming language is called **preservation**: given any well-typed expression e_0, if e_0 has type τ_0, then it must evaluate to a value of the same type τ_0. In this problem, you will show that a programming language with the rule FunWrong violates this property. You'll need (some of) the following typing rules.

$$\frac{\mathsf{n} \text{ is an integer literal}}{\Gamma \vdash \mathsf{n} : \texttt{int}}\ \textsc{intLit} \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \textsc{true} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{var}$$

$$\frac{\Gamma \vdash e_1 : \tau_2\ \texttt{->}\ \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ \textsc{app}$$

And you'll need (some of) the following semantics rules.

$$\frac{\mathsf{n} \text{ is an integer literal for } n}{\mathsf{n} \Downarrow n}\ \textsc{intLitE} \qquad \frac{}{\texttt{true} \Downarrow \top}\ \textsc{trueE} \qquad \frac{}{\texttt{fun}\ x\ \texttt{->}\ e \Downarrow \lambda x.e}\ \textsc{funE}$$

$$\frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v_2 \qquad e' = [v_2/x]e \qquad e' \Downarrow v}{e_1\ e_2 \Downarrow v}\ \textsc{appE}$$

Determine an expression e_0 such that $\{\} \vdash e_0 : \texttt{bool}$ and $e_0 \Downarrow 42$ according to the above rules (including funWrong). You must provide:

A. a typing derivation of $\{\} \vdash e_0 : \texttt{bool}$ and

B. a semantic derivation of $e_0 \Downarrow 42$.

$e_0 = (\text{fun } x \to x)\ 42$

A.

$$\dfrac{\dfrac{\dfrac{}{\{x : \text{bool}\} \vdash x : \text{bool}}\ (\text{var})}{\{\} \vdash \text{fun } x \to x : \text{int} \to \text{bool}}\ (\text{fun}) \qquad \dfrac{}{\{\} \vdash 42 : \text{int}}\ (\text{intLit})}{\{\} \vdash (\text{fun } x \to x)\ 42 : \text{bool}}\ (\text{app})$$

The problem is repeated here for convenience.

$$\frac{\mathsf{n} \text{ is an integer literal}}{\Gamma \vdash \mathsf{n} : \texttt{int}}\ \textsc{intLit} \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \textsc{true} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{var}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \texttt{ -> } \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ \textsc{app} \quad \frac{\mathsf{n} \text{ is an integer literal for } n}{\mathsf{n} \Downarrow n}\ \textsc{intLitE} \qquad \frac{}{\texttt{true} \Downarrow \top}\ \textsc{trueE}$$

$$\frac{}{\texttt{fun } x \texttt{ -> } e \Downarrow \lambda x.e}\ \textsc{funE} \qquad \frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v_2 \qquad e' = [v_2/x]e \qquad e' \Downarrow v}{e_1\ e_2 \Downarrow v}\ \textsc{appE}$$

Determine an expression e_0 such that $\{\} \vdash e_0 : \texttt{bool}$ and $e_0 \Downarrow 42$ according to the above rules (including FUNWRONG). You must provide:

A. a typing derivation of $\{\} \vdash e_0 : \texttt{bool}$ and

B. a semantic derivation of $e_0 \Downarrow 42$.

B.

$$\dfrac{\dfrac{}{\text{fun } x \rightarrow x \Downarrow \lambda x.x}\,(\text{funE}) \qquad \dfrac{}{42 \Downarrow 42}\,(\text{intLitE}) \qquad \dfrac{}{42 \Downarrow 42}\,(\text{intLitE})}{(\text{fun } x \rightarrow x)\ 42 \Downarrow 42}\,(\text{appE})$$

2 OCaml Programming

Implement the function `val lookup : ('a * 'b) list -> 'a -> 'b option` so that `lookup l a` is

- `Some b` if there is pair of the form `(a, b)` in `l`. If there are multiple such pairs, the output should be the value associated with the *rightmost* such pair in `l`;
- `None` if there is no such pair.

For example, the following assertions hold.

```
let _ = assert (lookup [] 2 = None)
let _ = assert (lookup [(2, "2")] 2 = Some "2")
let _ = assert (lookup [(2, "2"); (3, "3"); (2, "4"); (3, "5")] 2 = Some "4")
let _ = assert (lookup [(2, "2"); (3, "3"); (2, "4"); (3, "5")] 4 = None)
```

You *cannot* use anything from the standard library except for comparison operators and constructors, e.g., `(::)` and `[]`.

```
let lookup l a =
  let rec loop acc l =
    match l with
    | [] -> acc
    | (x, b) :: rest ->
      if x = a
      then loop (Some b) rest
      else loop acc rest
  in loop None l
```

3 Algebraic Data Types

Determine the smallest type `stuff` such that the following function is well-typed.

```
let do_thing (s : 'a stuff) (x : int) =
  match s with
  | Foo f -> f x
  | Bar (y, z) -> (if y then y else y) :: z
  | Baz l -> l.first (l.second (l.third = x))
```

```
type 'a stuff =
  | Foo of (int -> bool list)
  | Bar of bool * bool list
  | Baz of {
      first : 'a -> bool list;
      second : bool -> 'a;
      third : int
    }
```

4 Optional Binding

The programming language Swift has a feature called *optional binding*, which allows us to conditionally bind the inner value of an option. In this problem, we'll be looking at typing and semantic rules for a variant of this. We introduce `let?`-expressions, a version of `let`-expressions specialized to options. They have the following syntax:

<expr> ::= let? <expr> = <expr> in <expr>

Make sure to read the following rules carefully. The new `let?`-expressions are similar to `let`-expressions, but not identical.

A. Consider the following typing rules.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{var} \qquad \frac{\mathsf{n}\text{ is an integer literal}}{\Gamma \vdash \mathsf{n} : \texttt{int}}\ \textsc{intLit} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}\ \textsc{addInt}$$

$$\frac{}{\Gamma \vdash \texttt{None} : \tau\ \texttt{option}}\ \textsc{none} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{Some}\ e : \tau\ \texttt{option}}\ \textsc{Some}$$

$$\frac{\Gamma \vdash e_1 : \tau'\ \texttt{option} \qquad \Gamma, x : \tau' \vdash e_2 : \tau\ \texttt{option}}{\Gamma \vdash \texttt{let?}\ x \texttt{ = } e_1 \texttt{ in } e_2 : \tau\ \texttt{option}}\ \textsc{letOpt}$$

Write a derivation of the following typing judgment.

$$\{\} \vdash \texttt{let? x = Some 2 in Some (x + 2)} : \texttt{int option}$$

$$\dfrac{\dfrac{\dfrac{}{\{\} \vdash 2 : int}\ (intLit)}{\{\} \vdash Some\ 2 : int\ option}\ (some) \qquad \dfrac{\dfrac{\dfrac{}{\{x : int\} \vdash x : int}\ (var) \qquad \dfrac{}{\{x : int\} \vdash 2 : int}\ (intLit)}{\{x : int\} \vdash x + 2 : int}\ (addInt)}{\{x : int\} \vdash Some\ (x + 2) : int\ option}\ (some)}{\{\} \vdash let?\ x = Some\ 2\ in\ Some\ (x+2) : int\ option}\ (letOpt)$$

B. Consider the semantic rules. Note that we introduce an option value for the result of evaluating an option.

$$\dfrac{\mathsf{n} \text{ is an integer literal for } n}{\mathsf{n} \Downarrow n}\ \textsc{intLitE} \qquad \dfrac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 + v_2}{e_1 \texttt{ + } e_2 \Downarrow v}\ \textsc{addIntE}$$

$$\dfrac{}{\texttt{None} \Downarrow \mathsf{None}}\ \textsc{noneE} \qquad \dfrac{e \Downarrow v}{\texttt{Some } e \Downarrow \mathsf{Some}(v)}\ \overset{\text{Some E}}{\cancel{\textsc{noneE}}}$$

$$\dfrac{e_1 \Downarrow \mathsf{Some}(v_1) \qquad e' = [v_1/x]e_2 \qquad e' \Downarrow v}{\texttt{let? } x \texttt{ = } e_1 \texttt{ in } e_2 \Downarrow v}\ \textsc{letOptSome} \qquad \dfrac{e_1 \Downarrow \mathsf{None}}{\texttt{let? } x \texttt{ = } e_1 \texttt{ in } e_2 \Downarrow \mathsf{None}}\ \textsc{letOptNone}$$

Determine the value v such that the following semantic judgment is derivable, and then write its derivation.

$$\texttt{let? x = Some 2 in let? y = None in Some (x + y)} \Downarrow v$$

$$\dfrac{\dfrac{\dfrac{}{2 \Downarrow 2}\ (\text{intLitE})}{\text{Some } 2 \Downarrow \text{Some}(2)}\ (\text{someE}) \qquad \dfrac{\dfrac{}{\text{None} \Downarrow \text{None}}\ (\text{noneE})}{\text{let ? } y = \text{None in Some}(2+y) \Downarrow \text{None}}\ (\text{LON})}{\text{let? } x = \text{Some } 2 \text{ in let? } y = \text{None in Some } (x+y) \Downarrow \text{None}}\ (\text{LOS})$$