

CS242 Midterm

Fall 2021

- Please read all instructions (including these) carefully.
- There are 4 questions on the exam, some with multiple parts. You have 90 minutes to work on the exam.
- The exam is open note. You may use laptops, phones and e-readers to read electronic notes, but not for computation or access to the internet for any reason.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. Do not write on the back of exam pages or other pages.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

NAME: _____

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: _____

| Problem | Max points | Points |
|---------|------------|--------|
| 1 | 25 | |
| 2 | 10 | |
| 3 | 30 | |
| 4 | 25 | |
| TOTAL | 90 | |

1. **Lambda Calculus** (25 points)

Recall the following definitions of booleans, integers, and lists from homework 1:

```
def T =  $\lambda x. \lambda y. x$ ;
def F =  $\lambda x. \lambda y. y$ ;
def not =  $\lambda b. b \text{ F } T$ 
def inc =  $\lambda n. \lambda f. \lambda x. f (n f x)$ ;
def 0 =  $\lambda f. \lambda x. x$ ;
def cons =  $\lambda h. \lambda t. \lambda f. \lambda x. f h (t f x)$ ;
def nil =  $\lambda f. \lambda x. x$ ;
```

For the following problems, feel free to define and use helper functions.

- (a) Write a lambda expression `xor` that returns the `xor` of two booleans. Recall that `xor` is true iff exactly one of its two boolean arguments is true.

```
def xor =  $\lambda a. \lambda b. \underline{\hspace{4cm}}$ 
```

```
def xor =  $\lambda a. \lambda b. a (\text{not } b) b$ ;
```

- (b) Write a lambda expression `num_odds` that returns the number of odd integers in a list of integers.

```
def num_odds =  $\lambda l. \underline{\hspace{4cm}}$ 
```

```
def is_odd =  $\lambda n. n \text{ not } F$ ;
def num_odds =  $\lambda l. l (\lambda n. \lambda c. (\text{is\_odd } n) (\text{inc } c) c) 0$ ;
```

2. Object Calculus (10 points)

Consider the following object calculus, extended with some constants. Objects are defined by

$$o = [\dots, l_i : \zeta(x) e_i, \dots]$$

where the method bodies e_i can use variables x , integers i , sums of integers $e + e'$, method selections $e.l$, and method overrides $e.l \Leftarrow \zeta(y) e'$. Consider the following object calculus program:

$$o = [a : \zeta(x) 0, b : \text{_____}]$$

Fill in the **b** field with a method that, when invoked, returns an object of the same kind with the **a** field changed to a method that returns the old value of **a** incremented by 1. For example, the expression $o.b.b.b.b.b.a$ should evaluate to 5 (that is the number 5, not an encoding of 5).

$$o = [a : \zeta(x) 0, b : \zeta(y) y.a \Leftarrow \zeta(x) y.a + 1]$$

3. Simple Types (30 points)

In this problem we extend the Simply Typed Lambda Calculus (STLC) with some new features. As a reminder, STLC has no polymorphic (quantified) types. The STLC's syntax is:

$$\begin{aligned} \text{Constants } c &::= 0 \mid 1 \mid \dots \\ \text{Expressions } e &::= x \mid \lambda x : \tau. e \mid e e \mid c \\ \text{Types } \tau &::= \text{int} \mid \tau \rightarrow \tau \mid \alpha \end{aligned}$$

Recall that the type checking rules $\Gamma \vdash e : \tau$ for the STLC are:

$$\frac{}{\Gamma \vdash c : \text{int}} \quad \frac{}{\Gamma, x : \tau \vdash x : \tau} \quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau'} \quad \frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

- (a) We add *product* types to the STLC. Product types provide a way to construct and use pairs by packing two values together into a single value. Here we are introducing pairs as a new primitive type instead of using an encoding in the lambda calculus itself. Pairs are formed by the constructor $\langle e_1, e_2 \rangle$, and are eliminated through the accessors $.l$ and $.r$. For example $\langle 3, 4 \rangle.r \rightarrow 4$ and $\langle 3, 4 \rangle.l \rightarrow 3$.

A pair of expressions $\langle e_1, e_2 \rangle$ has type $\tau_1 \times \tau_2$ if $e_1 : \tau_1$ and $e_2 : \tau_2$. We extend the syntax of the STLC as follows:

$$\begin{aligned} \text{Expressions } e &::= \dots \mid \langle e, e \rangle \mid e.l \mid e.r \\ \text{Types } \tau &::= \dots \mid \tau \times \tau \end{aligned}$$

Fill in the typing rules for pairs:

$$\frac{\boxed{} \quad \boxed{}}{\Gamma \vdash \langle e_1, e_2 \rangle : \boxed{}}$$

$$\frac{\boxed{}}{\Gamma \vdash e.l : \boxed{}}$$

$$\frac{\boxed{}}{\Gamma \vdash e.r : \boxed{}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.l : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.r : \tau_2}$$

- (b) Consider the following approach to adding pairs to the STLC by using an encoding instead of adding them as primitives to the language:

```

let pair =  $\lambda x. \lambda y. \lambda f. f \ x \ y$ 
let dotl =  $\lambda p. p \ (\lambda x. \lambda y. x)$ 
let dotr =  $\lambda p. p \ (\lambda x. \lambda y. y)$ 
let p = pair 0 ( $\lambda f. f$ )
let x = dotl p
let y = dotr p

```

Is this program well typed? If so, provide the types of **pair**, **dotl** and **dotr**. If not, explain why the program cannot be typed.

The program is not well-typed, because the third argument to **pair** cannot be both $int \rightarrow (\alpha \rightarrow \alpha) \rightarrow int$ and $int \rightarrow (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$.

4. Combinator Calculi (25 points)

- (a) Show that the I combinator in the SKI calculus isn't needed: Write a combinator using only S and K that implements I. There is a solution that uses three combinators. S K K

- (b) Consider the following combinator language:

- $\langle x_1, \dots, x_n \rangle$ is a list of length n
- $\text{fold } f \ y \ \langle x_1, \dots, x_n \rangle = f \ x_1 \ (f \ x_2 \ (\dots (f \ x_n \ y) \dots))$
- $\text{cons } x_0 \ \langle x_1, \dots, x_n \rangle = \langle x_0, x_1, \dots, x_n \rangle$
- $\langle \rangle$ is the empty list, so we don't need a separate empty list constructor

Note that **fold** performs a reduction starting from the right end of the list using y as the initial value. In the following problems, you may use **fold**, **cons** and lambda expressions (including helper function definitions if you like) in your solution, but you may not use any form of iteration or recursion other than **fold**.

- i. Write a function *append* that appends an element x to the right end of a list l .
 $\text{append } x \ l = \text{fold cons } \langle x \rangle \ l$

- ii. Using *append*, write a function *reverse* that reverses a list l .
 $\text{reverse } l = \text{fold append } \langle \rangle \ l$