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:

$$\begin{aligned}
&\textsf{def T} = \lambda x.\ \lambda y.\ x;\\
&\textsf{def F} = \lambda x.\ \lambda y.\ y;\\
&\textsf{def not} = \lambda b.\ b\ \textsf{F T}\\
&\textsf{def inc} = \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x);\\
&\textsf{def 0} = \lambda f.\ \lambda x.\ x;\\
&\textsf{def cons} = \lambda h.\ \lambda t.\ \lambda f.\ \lambda x.\ f\ h\ (t\ f\ x);\\
&\textsf{def nil} = \lambda f.\ \lambda x.\ x;
\end{aligned}$$

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.

$\textsf{def xor} = \lambda a.\ \lambda b.$____________________________

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

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

$\textsf{def num_odds} = \lambda l.$____________________________

$\textsf{def is\_odd} = \lambda n.\ n\ \textsf{not}\ F;$

$\textsf{def num\_odds} = \lambda l.\ l\ (\lambda n.\ \lambda c.(\textsf{is\_odd}\ n)\ (\textsf{inc}\ c)\ c)\ \textsf{0};$

2. **Object Calculus** (10 points)
Consider the following object calculus, extended with some constants. Objects are defined by

$$\texttt{o} = [\ldots, \texttt{l}_\texttt{i} : \zeta(x)\ e_i, \ldots]$$

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:

$$\texttt{o} = [\texttt{a} : \zeta(x)\ 0,\ \texttt{b} : \underline{\hspace{6cm}}]$$

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).

$$\texttt{o} = [\texttt{a} : \zeta(x)0,\ \texttt{b} : \zeta(y)y.a <= \zeta(x)y.a + 1]$$

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:

$$\begin{array}{rrcl} \textit{Constants} & c & ::= & 0 \mid 1 \mid \ldots \\ \textit{Expressions} & e & ::= & x \mid \lambda x : \tau . e \mid e\ e \mid c \\ \textit{Types} & \tau & ::= & \mathsf{int} \mid \tau \to \tau \mid \alpha \end{array}$$

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

$$\frac{}{\Gamma \vdash c : \mathsf{int}} \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau . e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

(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 $.\mathsf{l}$ and $.\mathsf{r}$. For example $\langle 3, 4 \rangle.\mathtt{r} \to 4$ and $\langle 3, 4 \rangle.\mathtt{l} \to 3$.

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:

$$\begin{array}{rrcl} \textit{Expressions} & e & ::= & \ldots \mid \langle e, e \rangle \mid e.\mathsf{l} \mid e.\mathsf{r} \\ \textit{Types} & \tau & ::= & \ldots \mid \tau \times \tau \end{array}$$

Fill in the typing rules for pairs:

$$\frac{\square \qquad \square}{\Gamma \vdash \langle e_1, e_2 \rangle : \square}$$

$$\frac{\square}{\Gamma \vdash e.\mathsf{l} : \square}$$

$$\frac{\square}{\Gamma \vdash e.\mathsf{r} : \square}$$

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

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

$$\begin{aligned}
&\mathsf{let\ pair} = \lambda x.\ \lambda y.\ \lambda f.\ f\ x\ y \\
&\mathsf{let\ dotl} = \lambda p.\ p\ (\lambda x.\ \lambda y.\ x) \\
&\mathsf{let\ dotr} = \lambda p.\ p\ (\lambda x.\ \lambda y.\ y) \\
&\mathsf{let\ p} = \mathsf{pair}\ 0\ (\lambda f.\ f) \\
&\mathsf{let\ x} = \mathsf{dotl\ p} \\
&\mathsf{let\ y} = \mathsf{dotr\ p}
\end{aligned}$$

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 $int \to (\alpha \to \alpha) \to int$ and $int \to (\alpha \to \alpha) \to (\alpha \to \alpha)$.

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:

- $\langle x_1, \ldots, x_n \rangle$ is a list of length n
- `fold` $f\ y\ \langle x_1, \ldots, x_n \rangle = f\ x_1\ (f\ x_2\ (...(f\ x_n\ y)...))$
- `cons` $x_0\ \langle x_1, \ldots x_n \rangle = \langle x_0, x_1, \ldots, x_n \rangle$
- $\langle\rangle$ 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.
append $x\ l$ = `fold cons` $\langle x \rangle\ l$

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