CS242 Midterm
Fall 2024

- Please read all instructions (including these) carefully.
- There are 3 questions on the exam, all with multiple parts. You have 80 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.
- Write your answers in the boxes provided on the exam. We will grade what is in the answer boxes.
- 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	20	
2	20	
3	16	
TOTAL	56	

1. **Combinator Calculus** (20 points)

Recall the definitions of the combinators S, K, and I (lecture 2, slide 4):

$$\begin{aligned} I\ x &\longrightarrow x \\ K\ x\ y &\longrightarrow x \\ S\ x\ y\ z &\longrightarrow (x\ z)\ (y\ z) \end{aligned}$$

and that the abstraction algorithm $\mathcal{A}$ is defined as follows (lecture 2, slide 29):

$$\begin{aligned} \mathcal{A}(x, x) &= I \\ \mathcal{A}(E, x) &= K\ E \quad \text{if } x \text{ does not appear in } E \\ \mathcal{A}(E_1\ E_2, x) &= S\ \mathcal{A}(E_1, x)\ \mathcal{A}(E_2, x) \end{aligned}$$

In this problem, we consider variants of the SKI calculus and derive some of their properties.

(a) Consider replacing K with the following M combinator (M for "middle"):

$$M\ x\ y\ z \longrightarrow y$$

We first show that the SMI calculus is equivalent to the SKI calculus:

i. Express M in the SKI calculus (i.e., as a combinator composed of S, K, and I).

$M =$ smallest expression: K K
from using abstraction: K (S (K K) I)

ii. Express K in the SMI calculus (i.e., as a combinator composed of S, M, and I).

$K =$ smallest expression: M I

iii. Let $\mathcal{A}_M$ be the abstraction algorithm for the SMI calculus (i.e., $\mathcal{A}_M(E, x)\ x = E$). Fill in the following missing case.

$$\mathcal{A}_M(x, x) = I$$

$$\mathcal{A}_M(E, x) = \boxed{\text{(M I) E}}$$

if x does not appear in E

$$\mathcal{A}_M(E_1\ E_2, x) = S\ \mathcal{A}_M(E_1, x)\ \mathcal{A}_M(E_2, x)$$

(b) Now let's replace K by the following L combinator (L for "left"):

$$L\ x\ y\ z\ \longrightarrow x$$

We'll also show that the SLI calculus is equivalent to the SKI calculus:

i. Express L in the SKI calculus (i.e., as a combinator composed of S, K, and I).

$L =$ smallest expression: S (K K) K
from using abstraction: ((S (K K)) ((S (K K)) I))

ii. Express K in the SLI calculus (i.e., as a combinator composed of S, L, and I).

$K =$ smallest expression: ((S L) L)

(c) **Extra Credit:** (5 points)
Do not attempt this problem unless you have finished the rest of the exam!
Initially, it may appear that argument order doesn't have any impact on the SKI calculus as both M and L lead to equivalent calculi, but it actually turns out that if we replace K instead with a combinator R (R for "right"):

$$R\ x\ y\ z \longrightarrow z$$

that we actually get a calculus that is not equivalent to the SKI calculus! To show this result, prove that K cannot be expressed in the SRI calculus (i.e., that there does not exist any combinator C in the SRI calculus such that $C\ x\ y \rightarrow^* x$).
Hint: Evaluate some SRI calculus expressions in tree form. Do you see a pattern?

Claim: for any expression $e \longrightarrow^* e'$ in the SRI calculus, the rightmost node of e is the same as the rightmost node of e'.

Proof: by structural induction on the rewrite rules. For any reductions not applied at a parent of the rightmost node, the rightmost node trivially stays the same, so all we have to consider is rewrites at parents of the rightmost node. There are three reductions that can occur: $I\ z \rightarrow z$, $R\ x\ y\ z \rightarrow z$, and $S\ x\ y\ z \rightarrow (x\ z)\ (y\ z)$. In each of these cases, we see that the rightmost node of the left hand side of the reduction (i.e., z) is the same as the rightmost node of the right hand size of the reduction (i.e., z).

Claim: There does not exist an SRI calculus expression K_R such that $K_R\ x\ y \rightarrow^* x$.

Proof: $K_R\ x\ y$ is by assumption an expression in the SRI calculus, and as such cannot change the rightmost node of the expression, but the definition of K_R does just that (changing y to x). Thus, no such combinator K_R can exist.

2. **Lambda Calculus** (20 points)

In this problem, we write part of an interpreter for the lambda calculus in the lambda calculus itself. For all parts of this problem, you may use the following definitions in your solutions:

$$\begin{aligned}
\texttt{tt} &= \lambda x.\ \lambda y.\ x \\
\texttt{ff} &= \lambda x.\ \lambda y.\ y \\
\texttt{and} &= \lambda x.\ \lambda y.\ x\ y\ \texttt{ff} \\
\texttt{0} &= \lambda f.\ \lambda x.\ x \\
\texttt{succ} &= \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x) \\
\texttt{pair} &= \lambda x.\ \lambda y.\ \lambda z.\ z\ x\ y \\
\texttt{fst} &= \lambda x.\ \lambda y.\ x \\
\texttt{snd} &= \lambda x.\ \lambda y.\ y \\
\texttt{is-zero} &= \lambda n.\ n\ (\lambda x.\ \texttt{ff})\ \texttt{tt} \\
\texttt{pred-helper} &= \lambda p.\ \texttt{pair}\ (p\ \texttt{snd})\ (\texttt{succ}\ (p\ \texttt{snd})) \\
\texttt{pred} &= \lambda n.\ (n\ \texttt{pred-helper}\ (\texttt{pair 0 0}))\ \texttt{fst} \\
\texttt{sub} &= \lambda n.\ \lambda m.\ m\ \texttt{pred}\ n \\
\texttt{eq} &= \lambda n.\ \lambda m.\ \texttt{and}\ (\texttt{is-zero}\ (\texttt{sub n m}))\ (\texttt{is-zero}\ (\texttt{sub m n}))
\end{aligned}$$

All of these combinators have been introduced previously in class or homeworks, with the exception of `eq`, which takes two natural numbers and returns a boolean, true if the numbers are equal and false otherwise. Recall the structure of the lambda calculus:

$$\text{Term} \quad e ::= x \mid \lambda x.\ e \mid e_1\ e_2$$

We will use an algebraic data type `Term` to represent lambda calculus terms. Instead of representing variables as strings, e.g. "x" or "y", we will represent them as natural numbers `Nat`.

$$\texttt{type Term} = \texttt{var Nat} \mid \texttt{abs Nat Term} \mid \texttt{app Term Term}$$

(a) Encode each constructor of the `Term` algebraic data type as a lambda term using the construction given in class (lecture 4, slide 31).

$\texttt{var} = \lambda n.\ \lambda f.\ \lambda g.\ \lambda h.\ f\ n$

$\texttt{abs} = \lambda n.\ \lambda e.\ \lambda f.\ \lambda g.\ \lambda h.\ g\ n\ (e\ f\ g\ h)$

$\texttt{app} = \lambda e_1.\ \lambda e_2.\ \lambda f.\ \lambda g.\ \lambda h.\ h\ (e_1\ f\ g\ h)\ (e_2\ f\ g\ h)$

(b) Define a function `subst` that takes a term e_1, a variable (natural number) n, and another term e_2, and performs substitution (lecture 4, slide 9), returning $e_1[n := e_2]$. You may assume that:

- All variables in e_1 and e_2 are distinct (you do not need to worry about name collisions between variables in e_1 and variables in e_2 when performing substitution).
- The variable n is not bound by a lambda abstraction in e_1 — i.e., no subterm of the form $\lambda n.\ e'$ appears inside e_1.

You should not need helper functions beyond those defined above and the three constructors `var`, `abs`, and `app`. If you wish to define additional functions you may, but new functions may only refer to previously defined functions—no recursive definitions are permitted.

$\texttt{subst} = \lambda e_1.\ \lambda n.\ \lambda e_2.\ e_1\ (\lambda m.\ (\texttt{eq}\ m\ n)\ e_2\ (\texttt{var}\ m))\ \texttt{abs}\ \texttt{app}$

3. **Typed Lambda Calculus** (16 points)

In this question we will work in the polymorphic lambda calculus and, in some parts, the simply typed subset.

$$\begin{aligned} \text{Type} \quad & t ::= \alpha \mid t \to t \mid \texttt{Int} \\ \text{Quantified Type} \quad & o ::= \forall \alpha.\ o \mid t \\ \text{Term} \quad & e ::= x \mid \lambda x.\ e \mid e_1\ e_2 \mid \texttt{let}\ f = \lambda x.\ e_1\ \texttt{in}\ e_2 \mid i \end{aligned}$$

Recall the type rules of the simply typed lambda calculus with integers (lecture 5, slide 22):

$$\frac{}{A \vdash i : \texttt{Int}} \quad \text{(Int)}$$

$$\frac{}{A, x : t \vdash x : t} \quad \text{(Var)}$$

$$\frac{A, x : t \vdash e : t'}{A \vdash \lambda x.\ e : t \to t'} \quad \text{(Abs)}$$

$$\frac{A \vdash e : t \to t' \quad A \vdash e' : t}{A \vdash e\ e' : t'} \quad \text{(App)}$$

Also recall the additional rules for the polymorphic lambda calculus (lecture 6, slides 10 and 12):

$$\frac{A \vdash \lambda x.\ e : t \quad A, f : \forall \alpha_1, \ldots, \alpha_n.\ t \vdash e' : t' \quad \alpha_1, \ldots, \alpha_n \notin \mathrm{FV}(A)}{A \vdash \texttt{let}\ f = \lambda x.\ e\ \texttt{in}\ e' : t'} \quad \text{(Let)}$$

$$\frac{}{A, f : \forall \alpha.\ t \vdash f : t[\alpha := t']} \quad \text{(Inst)}$$

Fill in a valid type or write *untypable* if there is no way to type the term.

(a) $((\lambda x.\ x)\ (\lambda x.\ x))\ \lambda x.\ x$: $\alpha \to \alpha$

(b) $(\lambda f.\ \lambda g.\ f\ g)\ 0\ (\lambda x.\ x)$: untypable

(c) $(\lambda f.\ \lambda g.\ g\ f)\ 0\ (\lambda x.\ x)$: `Int`

(d) $(\lambda f.\ \lambda g.\ g\ f\ f)\ (\lambda x.\ x)\ (\lambda x.\ x)$: untypable

(e) $\texttt{let}\ f = \lambda x.\ x\ \texttt{in}\ (f\ f)\ f$: $\alpha \to \alpha$

(f) $\texttt{let}\ f = \lambda x.\ x\ \texttt{in}\ (\lambda g.\ g\ g)\ f$: untypable

(g) $\texttt{let}\ f = \lambda x.\ x\ \texttt{in}\ (f\ (\lambda x.\ x))\ (f\ 0)$: `Int`

(h) $\texttt{let}\ f = \lambda x.\ x\ \texttt{in}\ (\lambda f.\ (f\ (\lambda x.\ x))\ (f\ 0))\ f$: untypable