# Mid-term Exam

Foundations of Software

November 13, 2019

Last Name : ____________________

First Name : ____________________

Section : ____________________

Sciper : ____________________

| **Exercise** | **Points** | **Achieved Points** |
|---:|---:|---|
| 1 | 10 | |
| 2 | 10 | |
| 3 | 10 | |
| **Total** | 30 | |

## Exercise 1 : Strong Normalization of Linear STLC (10 points)

The linear lambda calculus (LLC) is a restriction of lambda calculus where every lambda-bound variable is required to be used *exactly* once. For instance, $\lambda x : \mathtt{S}.\, x$ and $\lambda x : \mathtt{S} \to \mathtt{T}.\, \lambda y : \mathtt{S}.\, x\ y$ are both valid terms in LLC, but $\lambda x : \mathtt{S} \to \mathtt{S} \to \mathtt{T}.\, \lambda y : \mathtt{S}.\, x\, y\, y$ is not (variable $y$ is used twice).

We can enforce linearity by rewriting the typing rules of STLC as follows:

$$x : \mathtt{T} \vdash x : \mathtt{T} \qquad \text{(T-Var)}$$

$$\frac{\Gamma,\, x : \mathtt{T}_1,\, \Gamma' \vdash t_2 : \mathtt{T}_2}{\Gamma,\, \Gamma' \vdash (\lambda x : \mathtt{T}_1.\ t_2)\ :\ \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma_1 \vdash t_1 : \mathtt{T}_1 \to \mathtt{T}_2 \quad \Gamma_2 \vdash t_2 : \mathtt{T}_1}{\Gamma_1,\, \Gamma_2 \vdash t_1\ t_2\ :\ \mathtt{T}_2} \qquad \text{(T-App)}$$

In a way, this system treats variables in $\Gamma$ like resources, which have to be consumed exactly once: notice that T-Var works only with the context $x : \mathtt{T}$, as opposed to T-Var in traditional lambda calculus, which works on contexts that can contain other bindings. Moreover, the rule for application T-App forces the current context to be split up into to distinct parts ($\Gamma_1$ and $\Gamma_2$), one for typing each subterm ($t_1$ and $t_2$) independently. Finally, the T-Abs rule allows inserting the lambda-bound variable at an arbitrary position in the current context (so it can be split correctly by T-App later).

The reduction rules for LLC are the same as for simply-typed lambda calculus (see the reference at the end).

For each of your proofs, you are expected to provide a rigorous formal argument, detailing every step of your proof and introducing lemmas if necessary.

### Question 1

**Prove the following lemma:**

**Lemma 1:** *If $\Gamma,\, x : S,\, \Gamma' \vdash t : T$ and $\vdash t' : S$ and $x$ does not apppear in $\Gamma$ nor $\Gamma'$, then $size([x \to t']t) \leq size(t) + size(t')$.*

Note that the definition of *size* is given in the reference of STLC. **To help with your proof, you can assume that the following lemma (Lemma 2) has already been proved:**

**Lemma 2:** *If $\Gamma \vdash t : T$ and $x$ does not appear in $\Gamma$, then $[x \to t']t = t$.*

## Solution to Question 1

**Proof of Lemma 1:** By induction on the structure of term $t$.
Base case: $t = y$, the only form without subterms, which can only be typed in context $y : \mathtt{T}$, so $y$ is $x$ and $[x \to t']t = t'$ which satisfies the hypothesis.
Inductive case:

- if $t = t_0\ t_1$, then the typing rule used must have been T-App, whose conclusion is (with the meta-variables renamed) $\Gamma_0, \Gamma_1 \vdash t_0\ t_1 : \mathtt{T}_2$. There are now two possibilities to consider: either the $x : \mathtt{S}$ binding appears in $\Gamma_0$ or it appears in $\Gamma_1$.
  - In the first case, let $\Gamma_0 = \Gamma_0', x : \mathtt{S}, \Gamma_0''$; we have, for some $\mathtt{U}$, $\Gamma_0', x : \mathtt{S}, \Gamma_0'' \vdash t_0 : \mathtt{U} \to \mathtt{T}$ and $\Gamma_1 \vdash t_1 : \mathtt{U}$. We use Lemma 2 to show that $[x \to t']t_1 = t_1$. Then, since $[x \to t']t = [x \to t']t_0\ [x \to t']t_1 = [x \to t']t_0\ t_1$ we conclude by IH that $\mathrm{size}([x \to t']t) = \mathrm{size}([x \to t']t_0\ [x \to t']t_1) = \mathrm{size}([x \to t']t_0) + \mathrm{size}(t_1) \leq \mathrm{size}(t_0) + \mathrm{size}(t') + \mathrm{size}(t_1) = \mathrm{size}(t_0\ t_1) + \mathrm{size}(t')$.
  - The other case is similar.
- if $t = \lambda y : \mathtt{U}.\, t_0$, then the typing rule used must have been T-Abs; we can apply the IH by noticing that $\mathrm{size}([x \to t']t) = \mathrm{size}(\lambda x : \mathtt{U}.\, [x \to t']t_0) = \mathrm{size}([x \to t']t_0) + 1 \leq \mathrm{size}(t_0) + \mathrm{size}(t') + 1 = \mathrm{size}(\lambda x : \mathtt{U}.\, t_0) + \mathrm{size}(t')$.

## Question 2

Our goal is to prove that LLC is strongly normalizing, meaning that reduction of terms typed with the rules above always terminates. To do so, we need to show that the size of the terms strictly decreases after each evaluation step, which means that reduction eventually stops on a normal form. **Your task is to prove that:**

*If* $\vdash t : \mathtt{T}$ *and* $t \to t'$*, then* $size(t') < size(t)$.

**To help with your proof, you can assume the lemmas of the previous exercises, as well as the following lemma:**

**Lemma (substitution):** if $x : \mathtt{S} \vdash t : \mathtt{T}$ and if $\vdash t' : \mathtt{S}$ then $\vdash [x \to t']t : \mathtt{T}$.

## Solution to Question 2

We prove that the size of the terms strictly decreases after each evaluation step, which means that reduction eventually stops on a normal form. We do this by induction on the structure of derivations of the evaluation relation $\longrightarrow$.

- if E-App1 or E-App2 was used last, then $t = t_1\ t_2$ and the rule used to type $t$ must have been T-App, which means that $\vdash t_1 : \mathtt{S} \to \mathtt{T}$ for some $\mathtt{S}$ and that $\vdash t_2 : \mathtt{S}$; by IH we conclude that size($t$) strictly decreases because either the size of $t_1$ does (in the case of E-App1) or the size of $t_2$ does (in the case of E-App2).

- if E-AppAbs was used last, we have $t = (\lambda x : \mathtt{S}.\, t_1)\ t_2$ for some $\mathtt{S}$ and $t \longrightarrow [x \to t_2]t_1$; the typing rule used for $t$ must have been T-App again so $\vdash t_1 : \mathtt{S} \to \mathtt{T}$ and $\vdash t_2 : \mathtt{S}$ and the rule used to type $(\lambda x : \mathtt{S}.\, t_1)$ must have been T-Abs; so we know that $x : \mathtt{S} \vdash t_1 : \mathtt{T}$ and by the substitution lemma we know that $\vdash [x \to t_2]t_1 : \mathtt{T}$, which lets us apply Lemma 1, proving $\text{size}([x \to t_2]t_1) \leq \text{size}(t_1) + \text{size}(t_2) < \text{size}(t_1) + \text{size}(t_2) + 1 = \text{size}((\lambda x : \mathtt{S}.\, t_1)\ t_2) = \text{size}(t)$.

## Exercise 2 : Encoding of Set (10 points)

In this exercise, we study the encoding of sets in *untyped lambda calculus*. The key idea is to encode a set as a fold operation. For intuition, below are Scala code that uses `foldLeft`:

```
def exists(set: Set, n: Int): Boolean =
    set.foldLeft(false) { (acc, e) => acc || e == n }

def length(set: Set): Int =
    set.foldLeft(0) { (acc, e) => acc + 1 }
```

You can assume that the elements of the set may be compared for equality with the following given function (in addition to lambda terms defined in the appendix):

- *eq x y*: whether the element $x$ and $y$ are equal, returns `tru` or `fls`.

As a hint, we may implement *length* based on the encoding of sets as follows:

$$length\ s = s\ c_0\ (\lambda acc.\lambda e.scc\ acc)$$

Your task is define the following terms:

1. *empty*: the empty set
2. *add*: return a new set with the given element added to a set
3. *remove*: return a new set with the given element removed from a set
4. *contains*: whether an element is in a set, returns `tru` or `fls`.

Note: make sure that the length of `add (add empty 0) 0` is 1.

```
empty = \z.\f.z

cons s x = \z.\f. s (f x z) f

remove s x = s empty (\acc.\e. (eq x e) acc (cons acc x))

add s x = \z.\f. (remove s x) (f x z) f

constains s x = s fls (\acc.\e. acc tru (eq e x))
```

## Exercise 3 : Track Usage of System Calls (10 points)

In this exercise we extend the Simply-Typed Lambda Calculus (STLC) to track the usage of *system calls* in programs. The goal is to be able to track in the type system system calls that may happen when evaluating a term.

In this language, we add natural numbers as well as the syntax for system call:

$$\begin{array}{rcll}
t & ::= & & \textbf{terms}: \\
& | & \ldots & \\
& | & n & \text{natural numbers} \\
& | & \texttt{call n t} & \text{system call} \\
n & ::= & 0, 1, 2, \ldots & \\
v & ::= & n \mid \lambda x{:}S.t &
\end{array}$$

In a system call *call $n$ $t$*, the number $n$ is the system call identifier, $t$ is the argument, which can be any term that type checks.

New evaluation rules:

$$\texttt{call}\ n\ v \longrightarrow \texttt{0} \qquad \text{(E-CallExec)}$$

$$\frac{t \longrightarrow t'}{\texttt{call}\ n\ t \longrightarrow \texttt{call}\ n\ t'} \qquad \text{(E-Call)}$$

To tell whether a function may perform system calls, we change the definition of types as follows:

$$\begin{array}{rcll}
T & ::= & & \textbf{types}: \\
& | & Nat & \text{natural numbers} \\
& | & T \xrightarrow{\delta} T & \text{function type} \\
\delta & \subset & \{\, 0, 1, 2, \ldots \,\} & \text{system call ids}
\end{array}$$

Calling a function of the type $T \xrightarrow{\emptyset} T$ will *never* perform any system call, while calling a function of the type $T \xrightarrow{\{1,2\}} T$ *may* perform system call 1 and/or 2, but will *never* perform system call 3.

The goal of this exercise is to define typing rules for STLC with the above extensions such that soundness holds for the system:

- (Progress) If $\emptyset \vdash t : T\ !\ \delta$, then either $t$ is a value or else $t \longrightarrow t'$.
- (Preservation) If $\Gamma \vdash t : T\ !\ \delta$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T\ !\ \delta'$ and $\delta' \subseteq \delta$.

The type system uses a typing judgment extended with a set of system call ids $\delta$, written $\Gamma \vdash t : T\ !\ \delta$. The set $\delta$ indicates the system calls that may be executed when evaluating the term $t$.

Your task is to define the typing rules for this extended language such that soundness holds. *You do not need to prove soundness.* Please make sure that the following typing judgments hold in your system:

- $\Gamma \vdash \lambda x{:}Nat.\texttt{call 1 x} : Nat \xrightarrow{\{1\}} Nat\ !\ \emptyset$.
- $\Gamma \vdash \lambda x{:}Nat.x : Nat \xrightarrow{\emptyset} Nat\ !\ \emptyset$.
- $\Gamma \vdash \lambda f{:}Nat \xrightarrow{\{1,2\}} Nat.f\ 0 : (Nat \xrightarrow{\{1,2\}} Nat) \xrightarrow{\{1,2\}} Nat\ !\ \emptyset$.

$$\Gamma \vdash n : Nat \, ! \, \emptyset \tag{T-Nat}$$

$$\frac{x \; : \; \mathtt{T} \in \Gamma}{\Gamma \vdash x \; : \; \; \mathtt{T} \, ! \, \emptyset} \tag{T-Var}$$

$$\frac{\Gamma, x : S \vdash t : T \, ! \, \delta}{\Gamma \vdash \lambda x{:}S.t : S \xrightarrow{\delta} T} \tag{T-Abs}$$

$$\frac{\Gamma \vdash t_1 : S \xrightarrow{\delta_3} T \, ! \, \delta_1 \qquad \Gamma \; ; \; E2 \vdash t_2 : S \, ! \, \delta_2}{\Gamma \vdash t_1 \; t_2 : T \, ! \, \delta_1 \cup \delta_2 \cup \delta_3} \tag{T-App}$$

$$\frac{\Gamma \vdash t : T \, ! \, \delta}{\Gamma \vdash \mathtt{call} \; n \; t : Nat \, ! \, \delta \cup \{n\}} \tag{T-Call}$$

## For reference: **Untyped lambda calculus**

The complete reference of the untyped lambda calculus with call-by-value semantics is:

$$
\begin{array}{rcll}
t & ::= & & \textbf{terms}: \\
 & | & x & \textit{variable} \\
 & | & \lambda x.\, t & \textit{abstraction} \\
 & | & t\ t & \textit{application (left assoc.)} \\
 & & & \\
v & ::= & & \textbf{values}: \\
 & | & \lambda x.\, t & \textit{abstraction}
\end{array}
$$

Small-step reduction rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(R-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(R-App2)}$$

$$(\lambda\ x.\ t_1)\ v_2 \longrightarrow [x \mapsto v_2]\ t_1 \qquad \text{(R-AppAbs)}$$

## For reference: **Predefined Lambda Terms**

Predefined lambda terms that can be used as-is in the exam

$$
\begin{array}{ll}
\texttt{unit} = & \lambda x.\ x \\
 & \\
\texttt{tru} = & \lambda t.\ \lambda f.\ t \\
\texttt{fls} = & \lambda t.\ \lambda f.\ f \\
\texttt{iszro} = & \lambda m.\ m\ (\lambda x.\ \texttt{fls})\ \texttt{tru} \\
\texttt{test} = & \lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f\ \texttt{unit} \\
 & \\
\texttt{pair} = & \lambda f.\ \lambda s.\ \lambda b.\ b\ f\ s \\
\texttt{fst} = & \lambda p.\ p\ \texttt{tru} \\
\texttt{snd} = & \lambda p.\ p\ \texttt{fls} \\
 & \\
\texttt{c}_0 = & \lambda s.\ \lambda z.\ z \\
\texttt{c}_1 = & \lambda s.\ \lambda z.\ s\ z \\
\texttt{scc} = & \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z) \\
\texttt{plus} = & \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z) \\
\texttt{times} = & \lambda m.\ \lambda n.\ m\ (\texttt{plus}\ n)\ c_0 \\
 & \\
\texttt{zz} = & \texttt{pair}\ c_0\ c_0 \\
\texttt{ss} = & \lambda p.\ \texttt{pair}\ (\texttt{snd}\ p)\ (\texttt{scc}\ (\texttt{snd}\ p)) \\
\texttt{prd} = & \lambda m.\ \texttt{fst}\ (m\ \texttt{ss}\ \texttt{zz})
\end{array}
$$

## For reference: **Simply Typed Lambda Calculus**

The complete reference of the simply typed lambda calculus is:

$$\begin{array}{llll}
t & ::= & & \textbf{terms}: \\
 & | & x & \textit{variable} \\
 & | & \lambda\, x\!:\!\texttt{T}.\ t & \textit{abstraction} \\
 & | & t\ t & \textit{application} \\
 & & & \\
v & ::= & & \textbf{values}: \\
 & | & \lambda\, x\!:\!\texttt{T}.\ t & \textit{abstraction-value} \\
 & & & \\
\texttt{T} & ::= & & \textbf{types}: \\
 & | & \texttt{T} \to \texttt{T} & \textit{type of functions (right assoc.)}
\end{array}$$

Evaluation rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

$$(\lambda\ x\!:\ \texttt{T}_1.\ t_1)\ v_2 \longrightarrow [x \to v_2]\ t_1 \qquad \text{(E-AppAbs)}$$

Typing rules:

$$\frac{x\ :\ \texttt{T} \in \Gamma}{\Gamma \vdash x\ :\ \texttt{T}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma,\ x\ :\ \texttt{T}_1 \vdash t_2\ :\ \texttt{T}_2}{\Gamma \vdash (\lambda\ x\!:\ \texttt{T}_1.\ t_2)\ :\ \texttt{T}_1 \to \texttt{T}_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1\ :\ \texttt{T}_1 \to \texttt{T}_2 \quad \Gamma \vdash t_2\ :\ \texttt{T}_1}{\Gamma \vdash t_1\ t_2\ :\ \texttt{T}_2} \qquad \text{(T-App)}$$

Definition of *size*:

$$\begin{aligned}
\text{size}(x) &= 1 \\
\text{size}(\lambda x : \texttt{T}.\, t) &= \text{size}(t) + 1 \\
\text{size}(t_1\ t_2) &= \text{size}(t_1) + \text{size}(t_2)
\end{aligned}$$