Your solutions to this assignment should be typeset using LaTeX or Word. If you use LaTeX, there is a template available for your use at the course website. (Remember to look in b522.sty for macros you can use.) A pdf file containing the solutions should be submitted by 11:59pm on the due date.

Note: This homework is worth 120 points. It is a longer and more difficult assignment than the last one, so plan your time accordingly.

1. **Well-founded relations** (25 pts.)

   Which of the following relations are well-founded? Briefly explain why or why not.

   (a) Dictionary ordering on strings of alphabetic characters (a-z).

   (b) An ordering $\prec$ on pairs of natural numbers defined inductively by these rules:

   $$\frac{n_1 < n_1'}{(n_1, n_2) \prec (n_1', n_2)} \qquad \frac{n_2 < n_2'}{(n_1, n_2) \prec (n_1, n_2')}$$

   (c) A relation $\prec$ on finite trees, where given two trees t and t', $t \prec t'$ iff t is exactly the same as t' except that it is missing exactly one leaf.

   (d) An ordering $\prec$ on finite sequences of natural numbers, where a sequence s of length n is preceded by the subsequences of s and also by any sequence whose first n elements are all smaller than the corresponding elements of s.

   (e) A relation $\prec$ on partial functions in $\mathbb{N} \rightharpoonup \mathbb{N}$, where

   $$f_1 \prec f_2 \stackrel{\triangle}{\iff} f_1 \neq f_2 \ \wedge\ \mathrm{dom}(f_1) \subseteq \mathrm{dom}(f_2) \ \wedge\ \forall x \in \mathrm{dom}(f_1).\, f_1(x) \leq f_2(x).$$

2. **Names and scope** (20 pts.)

   Consider the following program:

   ```
   let x = 5 in
     let f = λy. x + y in
       let x = 4 in
         let g = (λz. let x = 3 in f(x)) in
           g(x) + f(x)
   ```

   (a) What does this program output using call-by-value semantics with static scope? Explain briefly.

   (b) What does this program output using call-by-value semantics with dynamic scope? Explain briefly.

   (c) What does this program output using call-by-name semantics with static scope? Explain briefly.

3. **Call-by-denotation** (25 pts.)

   In class we saw two different ways of evaluating the free variables in function bodies: static scoping and dynamic scoping. Static scoping uses the environment of the function definition (the lexical scope), and dynamic scoping uses the environment of the function evaluation (the dynamic scope).

   A similar distinction can by made with the evaluation of the actual arguments of a function: we could evaluate the free variables of actual arguments using either the environment at the function application (the lexical scope) or the environment at the evaluation of the actual arguments (the dynamic scope).

In call-by-value semantics, since the actual arguments are evaluated before applying the function, the lexical and dynamic scope are the same. However, when the arguments are evaluated lazily, the distinction is important.

We use *call-by-denotation* to refer to lazy evaluation of the actual arguments using dynamic scope, where even the choice of scope to use is lazy—the environment used to look up the values of variables is the one in force when the variable's value is needed. (We continue to use call-by-name to mean lazy evaluation of the actual arguments using the static scope.) For example, consider the following program. Using call-by-denotation semantics, the program evaluates to 1; using call-by-name semantics it evaluates to 2.

$$\begin{array}{l}\mathsf{let}\ f = \lambda y.\, \mathsf{let}\ x = 0\ \mathsf{in}\ y\ \mathsf{in}\\ \quad \mathsf{let}\ x = 1\ \mathsf{in}\\ \quad\quad f(x\ +\ 1)\end{array}$$

The TeX language has a semantics similar to call-by-denotation. For example, the following TeX code results in the text "inside", because the macro `foo` isn't expanded until after it is redefined.

```
\def\fn#1{\def\foo{inside} #1}
\def\foo{outside}
\fn\foo
```

The corresponding OCaml code would be something like the following, which evaluates to "`outside`" in that language.

```
let fn = (fun x -> let foo = "inside" in x) in
  let foo = "outside" in
    fn foo
```

(a) What would the result of evaluating the following program be, using call-by-denotation semantics?

$$\begin{array}{l}\mathsf{let}\ x = 0\ \mathsf{in}\\ \quad \mathsf{let}\ f = \lambda y.\, x + y\ \mathsf{in}\\ \quad\quad \mathsf{let}\ x = 1\ \mathsf{in}\\ \quad\quad\quad f(x+1)\end{array}$$

(b) Give a translation of dynamically scoped call-by-denotation lambda calculus into statically scoped uML, analogously to the translations given in class for dynamic and static scoping. That is, the source language uses dynamic scope to evaluate free variables in function bodies, and free variables in function arguments. Briefly explain the key differences between this translation and the translation for statically scoped, eager evaluation.

4. **Dangling references** (50 pts.)

In class we claimed that during evaluation, uML! programs never generate dangling references. Let's prove it. Consider the fragment of uML! consisting of the following expressions and values:

$$\begin{array}{rcl} e & ::= & n \mid x \mid \mathbf{ref}\, e \mid\ !e \mid e_1 := e_2 \mid \mathbf{null} \mid \lambda x.\, e \mid e_1\ e_2 \mid \\ & & \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid (e_1,\, e_2) \mid \mathbf{let}\ (x, y) = e_1\ \mathbf{in}\ e_2 \\ v & ::= & n \mid (v_1,\, v_2) \mid \mathbf{null} \mid \lambda x.\, e \ \text{(where } \lambda x.\, e \text{ is closed)} \end{array}$$

To define the small-step semantics of uML!, we augment the grammar of expressions and values with a set of locations $\ell \in \mathbf{Loc}$.

$$\begin{array}{rcl} e & ::= & \ldots \mid \ell \\ v & ::= & \ldots \mid \ell \end{array}$$

A *store* σ is a partial map from locations to values (which could be other locations). The small-step semantics of uML! programs was defined in terms of *configurations* $\langle e, \sigma \rangle$, where e is an augmented

expression and σ is a store. (For your reference, the small-step operational semantics of uML! is given at the end of this document.)

We define $loc(e)$ to be the set of locations that occur in the expression e. Thus, for example, $loc((!\,\ell_2)\ (\lambda x.\,(!\,\ell_1)+!\,(\mathbf{ref}\ 4))) = \{\ell_1, \ell_2\}$.

A uML! *program* is a closed expression that does not contain any locations. Thus, if e is a program then $loc(e) = \emptyset$.

(a) Consider the following uML! configuration:

$$\langle\ (\lambda x.\,(!\,\ell_1)\ 2)\ (\mathbf{ref}\ 1)\ ,\ \{\ell_1 \mapsto \lambda y.\,\mathbf{ref}\ y\}\ \rangle$$

Show the evaluation of this configuration. For each configuration $\langle e', \sigma'\rangle$ in the evaluation, give $loc(e')$.

(b) Give an inductive definition of the set $loc(e)$ of locations occurring in e.

(c) Prove that if e is a uML! program and $\langle e, \emptyset\rangle \longrightarrow^* \langle e', \sigma\rangle$, then $loc(e') \subseteq \mathrm{dom}(\sigma)$. If you use induction, identify the relation you are using in your induction and argue that it is well-founded.

Small-Step Operational Semantics of uML!

Evaluation contexts

$$\begin{array}{rcl} E & ::= & [\cdot] \mid \mathbf{ref}\ E \mid !\,E \mid E := e_2 \mid v_1 := E \mid E\ e_2 \mid v_1\ E \mid \\ & & \mathbf{let}\ x = E\ \mathbf{in}\ e_2 \mid (E,\, e_2) \mid (v_1,\, E) \mid \mathbf{let}\ (x, y) = E\ \mathbf{in}\ e_2 \end{array}$$

Reductions

$$\begin{array}{rcll} \langle \mathbf{ref}\ v,\, \sigma\rangle & \longrightarrow & \langle \ell,\, \sigma[\ell \mapsto v]\rangle & (\text{where } \ell \notin \mathrm{dom}(\sigma)) \\ \langle !\,\ell,\, \sigma\rangle & \longrightarrow & \langle \sigma(\ell),\, \sigma\rangle & (\text{where } \ell \in \mathrm{dom}(\sigma)) \\ \langle \ell := v,\, \sigma\rangle & \longrightarrow & \langle \mathbf{null},\, \sigma[\ell \mapsto v]\rangle & (\text{where } \ell \in \mathrm{dom}(\sigma)) \\ \langle (\lambda x.\, e)\ v,\, \sigma\rangle & \longrightarrow & \langle e\{v/x\},\, \sigma\rangle & \\ \langle \mathbf{let}\ x = v\ \mathbf{in}\ e,\, \sigma\rangle & \longrightarrow & \langle e\{v/x\},\, \sigma\rangle & \\ \langle \mathbf{let}\ (x, y) = (v_1, v_2)\ \mathbf{in}\ e,\, \sigma\rangle & \longrightarrow & \langle e\{v_1/x\}\{v_2/y\},\, \sigma\rangle & \end{array}$$

Context rule

$$\frac{\langle e,\, \sigma\rangle \longrightarrow \langle e',\, \sigma'\rangle}{\langle E[e],\, \sigma\rangle \longrightarrow \langle E[e'],\, \sigma'\rangle}$$