CSE 526: Principles of Programming Languages

Spring 2010 **Final Exam** Max: 100 points

May 13, 2010 Duration: 2h 30m

---

1. An untyped lambda term t is typable if there is a typed lambda term t' such that $erase(t') = t$ and t' is well-typed. The definition of *erase*, taken from the textbook, is as follows:

$$\begin{array}{rcl} erase(x) & = & x \\ erase(\lambda x : T.\ t) & = & \lambda x.\ erase(t) \\ erase(t_1\ t_2) & = & erase(t_1)\ erase(t_2) \end{array}$$

   (a) [15 points] State whether each of the following untyped lambda terms is typable. If a term is typable, give its principal type. If a term is not typable, formally explain why not.

      i. $\lambda x.\ \lambda y.\ (x\ y)$
      ii. $\lambda x.\ (x\ x)$
      iii. $\lambda x.\ \lambda y.\ x\ (x\ y)$
      iv. $\lambda x.\ \lambda y.\ x\ (y\ x)$
      v. $\lambda x.\ \lambda y.\ (x\ y)\ y$

   (b) [8 points] Show that if t is typable, then every subterm of t is typable.

2. A lambda term is said to be follow the *De Barendregt convention* if the names of all bound variables are different from each other, and different from any free variable. Every lambda term can be alpha-converted to a term that follows the De Barendregt convention. Note that De Barendregt convention is different from the de Bruijn notation.

   (a) [6 points] For each of the following lambda terms, write an alpha-equivalent term that follows the De Barendregt convention.

      i. $\lambda x.\ (\lambda y.\ x\ y)\ y$
      ii. $\lambda x.\ \lambda y.\ x\ (y\ \lambda x.\ x)$
      iii. $\lambda x.\ (x\ x)$

   (b) [9 points] Formally define the De Barendregt convention using an inductive definition. More specifically, give an inductive definition of a function *followsdb* whose domain is the set of all lambda terms and whose range is Boolean, such that *followsdb* maps t to *true* iff t follows the De Barendregt convention. You may assume the definitions of the set of free variables of t (denoted by $FV(t)$) and the set of all variables of t (denoted by $Vars(t)$). If you need additional auxilliary definitions, make sure those are defined inductively too.

3. (a) [4 points] Recall the extension of pure lambda calculus with `let` construct of the form "`let` $x =$ t_1 `in` t_2". This construct can be considered as syntactic sugar, explanding to $(\lambda x.\ t_2)\ t_1$.
   OCAML's let construct is a more general and has the form `let` $x_1\ x_2\ \ldots x_n = t_1$ `in` t_2 where $x_1, x_2, \ldots, x_n$ are distinct variables. For instance, `let x = 2 in let f y = y+1 in f x` is an OCAML expression. Show OCAML's let expressions can be treated as syntactic sugar; i.e. expand OCAML's let expressions into pure lambda terms. [For full credit, your description should be precise and complete].

(b) [5 points] Another extension to the lambda calculus discussed in class is the `fix` construct to permit the definition of recursive functions. OCAML has the `let rec` construct for defining recursive functions. Show how OCAML expressions with `let rec` can be converted to lambda calculus with `fix`. [For full credit, your description should be precise and complete]

(c) [5 points] Show how mutually recursive functions can be written in lambda calculus; you may use any extension of the lambda calculus (including, but not necessarily: let, records, variants, fix, references and exceptions).

(d) [5 points] Consider the OCAML expression

```
let x = nil in (1::x, true::x)
```

Is the above expression well typed? Why or why not?

(e) [5 points] Consider the OCAML expression

```
(fun x -> (1::x, true::x)) nil
```

where "`fun x -> t`" is OCAML's way of writing $\lambda x.\ t$
Is the above expression well typed? Why or why not?

4. (a) [12 points] Write the normal forms of the following expressions in lambda calculus extended with let, tuples and references.

   i. $\texttt{let}\ r = \texttt{ref}\ 0\ \texttt{in}\ !r$
   ii. $\texttt{let}\ a = \{\texttt{ref}\ 0, \texttt{ref}\ 0\}\ \texttt{in}\ !(a.1)$
   iii. $\texttt{let}\ a = \{\texttt{ref}\ 0, \texttt{ref}\ 0\}\ \texttt{in}\ a.1 := 1; !(a.2)$
   iv. $\texttt{let}\ b = \texttt{let}\ x = \texttt{ref}\ 0\ \texttt{in}\ \{x, x\}\ \texttt{in}\ b.1 := 1; !(b.2)$

   (b) [12 points] Write expressions in lambda calculus extended with let, tuples and references that, when evaluated in an empty store result in the following stores.

   i. $\{l_1 \mapsto 0\}$
   ii. $\{l_1 \mapsto \{0, 0\}\}$
   iii. $\{l_1 \mapsto \{l_2, 1\}, l_2 \mapsto 0\}$
   iv. $\{l_1 \mapsto \lambda x : \texttt{Nat}.\ (!l_1)\ x\}$

5. [14 points] Let `MS <: Grad <: Student` be a subtype relation among base types `MS`, `Grad` and `Student`.

   (a) Which of the following are subtypes of `{a:Grad, b:Student}`? Justify each answer briefly.

   i. `{a:Grad}`
   ii. `{a:Student, b:MS}`
   iii. `{a:Grad, b:MS, c:Student}`

   (b) Which of the following are subtypes of `Grad` → `Grad`? Justify each answer briefly.

   i. `MS` → `Grad`
   ii. `Grad` → `MS`
   iii. `Grad` → `Student`
   iv. `Student` → `Grad`

END OF EXAM