CSCI B522 Programming Language Foundations

Spring 2003

Final

Maximum: 40 points (40% of Final Grade)

Name (please print): . . . . . .

Id: . . . . . .

1	Haskell	8 pts	
2	Types	8 pts	
3	Semantics I	8 pts	
4	Proofs I	8 pts	
5	Java Bytecode Language	8 pts	
6	Proofs II	8 pts	
Total		48 pts	

1 Haskell

There are four questions about Haskell:

1.1 Recursive Programming

You are given the following datatype of natural numbers with addition:

```
data Nat = Zero | Succ Nat

plus :: Nat -> Nat -> Nat
plus x Zero = x
plus x (Succ y) = Succ (plus x y)
```

Define the function `multiply :: Nat -> Nat -> Nat` with the obvious meaning.
Solution:

```
multiply :: Nat -> Nat -> Nat
multiply x y = rep (plus x) y

rep f Zero = Zero
rep f (Succ n) = f (rep f n)
```

1.2 List Manipulation

The definitions of `map`, `(++)` (append), `foldl`, and `foldr` from the standard library are:

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)
```

Without using any recursive calls, write the following functions:

- `sum :: (Num a) => [a] -> a` which returns the sum of a list of numbers:

  ```
  > sum [1,2,3,4]
  10
  ```

  Solution: `sum = foldr (+) 0`

- `product :: (Num a) => [a] -> a` which returns the product of a list of numbers:

  ```
  > product [1,2,3,4]
  24
  ```

  Solution: `product = foldr (*) 1`

- `concat :: [[a]] -> [a]` which takes a list of lists which it appends all together:

  ```
  > concat [[1,2,3], [4,5], [], [6,7]]
  [1,2,3,4,5,6,7]
  ```

  Solution: `concat = foldr (++) []`

1.3 Reading Haskell

What does `main` below return?

```
f (h:t) = h : f [ x | x <- t, x `mod` h /= 0]
main = f [2..]
```

Solution: All prime numbers

1.4 Typing Haskell

Consider the following program:

```
foldl f z [] = [z]
foldl f z (x:xs) = foldl f (f z x) xs

flip f x y = f y x

reverse = foldl (flip (\ x xs -> x : xs)) []

palin xs = reverse xs == xs
```

The intention is that `palin` would take a list and check if it is a palindrome. Unfortunately compiling this program produces the following message:

```
ERROR "Ex1.hs":8 - Type error in application
*** Expression     : reverse xs == xs
*** Term           : reverse xs
*** Type           : [[a]]
*** Does not match : [a]
*** Because        : unification would give infinite type
```

Identify the type error and correct it.
Solution: The first line should be `foldl f z [] = z`

2 Types

You are given the syntax and type system for the following small language. A type t is either an `int` or a function type. An expression e is either the constant K, the constant S, or an application of two expressions:

$$\begin{array}{rcl} t & ::= & \texttt{int} \mid t \to t \\ e & ::= & K \mid S \mid ee \end{array}$$

The type rules are:

$$\frac{}{\vdash K : t_1 \to (t_2 \to t_1)} \qquad \frac{}{\vdash S : (t_1 \to (t_2 \to t_3)) \to ((t_1 \to t_2) \to (t_1 \to t_3))}$$

$$\frac{\vdash e_1 : t_2 \to t \qquad \vdash e_2 : t_2}{\vdash e_1 e_2 : t}$$

Give a type derivation for the term: $((SK)K)$
Solution:

$$\frac{\dfrac{\vdash S : (t \to ((t' \to t) \to t)) \to ((t \to (t' \to t)) \to (t \to t)) \quad \vdash K : t \to ((t' \to t) \to t)}{\vdash SK : (t \to (t' \to t)) \to (t \to t)} \quad \vdash K : t \to (t' \to t)}{\vdash SKK : t \to t}$$

3 Semantics I

Complete the following interpreter for this very simplified call-by-value functional language:

```
data Exp =
   Var String               -- x
 | Fun String Exp           -- lambda x. e
 | App Exp Exp              -- e1 e2

data Value = Closure Exp Env
type Env = [(String,Value)]

eval :: Exp -> Env -> Value

eval (Var s) env =
  let binding = lookup s env
  in case binding of
       Just v -> v

eval (e @ (Fun x body)) env = ...




eval (App e1 e2) env = ...
```

Solution:

```
eval :: Exp -> Env -> Value
eval (Var s) env =
  let binding = lookup s env
  in case binding of
       Just v -> v
eval (e @ (Fun x body)) env = Closure e env
eval (App e1 e2) env =
    let Closure (Fun s body) env' = eval e1 env
        v2 = eval e2 env
    in eval body ((s,v2):env')
```

4 Proofs I

We define a binary tree as follows:

```
data tree = Leaf | Node tree tree
```

Here are three definitions that compute the size, the number of leaves, and the number of nodes in a tree:

```
size Leaf = 1
size (Node t1 t2) = size t1 + size t2 + 1

leaves Leaf = 1
leaves (Node t1 t2) = leaves t1 + leaves t2

nodes Leaf = 0
nodes (Node t1 t2) = nodes t1 + nodes t2 + 1
```

Prove the following theorem: for any binary tree `t`, we have that `size(t) = leaves(t) + nodes(t)`.
Solution: By induction on the structure of `t`:

- `t = Leaf`: we have that

  ```
  size Leaf = 1,
  leaves Leaf = 1, and
  nodes Leaf = 0.
  ```

  Check 1 = 1 + 0.

- `t = Node t_1 t_2`: we have that

  ```
  size (Node t1 t2) = size t1 + size t2 + 1
  leaves (Node t1 t2) = leaves t1 + leaves t2
  nodes (Node t1 t2) = nodes t1 + nodes t2 + 1
  ```

  Since `t1` and `t2` are smaller trees, the inductive hypothesis gives:

  ```
  size t1 = leaves t1 + nodes t1
  size t2 = leaves t2 + nodes t2
  ```

  Substituting and calculating we have:

  ```
  size (Node t1 t2)
   = size t1 + size t2 + 1
   = leaves t1 + nodes t1 + leaves t2 + nodes t2 + 1
   = (leaves t1 + leaves t2) + (nodes t1 + nodes t2 + 1)
   = leaves (Node t1 t2) + nodes (Node t1 t2)
  ```

5 Java Bytecode Language

According to the dynamic semantics (not the type systems) we studied and implemented, does this program evaluate without errors? (See Appendix for the rules.) If you answer that the program is correct, show the contents of the local variables and stack after each instruction. If you answer that the program is incorrect, explain the error.

(You need three local variables for this example; they are initialized to unusable values ◊; the stack is initially empty.)

PC	Instruction	Locals	Stack
1	*new* "C"	◊ ◊ ◊	
2	*store* 0	◊ ◊ ◊	(Uninit "C" 1 100)
3	*load* 0	(Uninit "C" 1 100) ◊ ◊	
4	*store* 1	(Uninit "C" 1 100) ◊ ◊	(Uninit "C" 1 100)
5	*load* 0	(Uninit "C" 1 100) (Uninit "C" 1 100) ◊	
6	*store* 2	(Uninit "C" 1 100) (Uninit "C" 1 100) ◊	(Uninit "C" 1 100)
7	*jsr* 11	(Uninit "C" 1 100) (Uninit "C" 1 100) (Uninit "C" 1 100)	
8	*load* 2	(Ret 8) (Obj "C" 100) (Obj "C" 100)	
9	*use* "C"	(Ret 8) (Obj "C" 100) (Obj "C" 100)	(Obj "C" 100)
10	*halt*	(Ret 8) (Obj "C" 100) (Obj "C" 100)	
11	*store* 0	(Uninit "C" 1 100) (Uninit "C" 1 100) (Uninit "C" 1 100)	(Ret 8)
12	*load* 1	(Ret 8) (Uninit "C" 1 100) (Uninit "C" 1 100)	
13	*init* "C"	(Ret 8) (Uninit "C" 1 100) (Uninit "C" 1 100)	(Uninit "C" 1 100)
14	*ret* 0	(Ret 8) (Obj "C" 100) (Obj "C" 100)	

6 Proofs II

In Java, the typing rule for conditional expressions `(b ? e1 : e2)` is as follows:

- The guard `b` must have type boolean
- if the expressions `e1` and `e2` have reference types `t1` and `t2`, then either:
  - `t1` and `t2` are identical, or
  - one of `t1` and `t2` must be a subtype of the other
- ...

For the purposes of this question, the semantics of Java is given by an abstract machine which includes among other rules the following rule for method calls:

$$m(a_1, a_2) \rightarrow r[a_1/x_1,\ a_2/x_2]$$

where m has the definition:

$$R\ m\ (t_1\ x_1,\ t_2\ x_2)\ \{\ \textbf{return}\ r;\ \}$$

and $r[a_1/x_1,\ a_2/x_2]$ is the expression r with all free occurrences of x_1 and x_2 replaced by a_1 and a_2.

It is known that type safety fails for the fragment of Java described above. Find a counterexample.

Hint: Consider the following method:

```
Object m (Object a1, Object a2) { return (true ? a1 : a2); }
```

Solution:

The proof of subject reduction fails as follows. Consider the following program fragment where classes `A` and `B` are unrelated:

```
Object m (Object a1, Object a2) { return (true ? a1 : a2); }
m(new A(), new B())
```

The above state typechecks. Let's do one evaluation step. We get:

```
Object m (Object a1, Object a2) { return (true ? a1 : a2); }
return true ? new A() : new B();
```

which no longer typechecks.