THE UNIVERSITY OF CALGARY

FACULTY OF SCIENCE

FINAL EXAMINATION

COMPUTER SCIENCE 417

April, 2002 Time: 2 hrs.

Instructions

The exam contains questions totalling 100 points. Answer all questions. This exam is closed book.

30 marks

1. In order to implement a type checker for the extended types λ calculus it is necessary to write a unification algorithm for terms. If we let `Term` be the following datatype:

```
data Term = Var Int
          | Node (String,[Term])
```

the aim of this question is that you should write a unification algorithm in Haskell

$$\texttt{unify} :: \texttt{Term}-> \texttt{Term}-> \texttt{Maybe}[(\texttt{Int},\texttt{Term})],$$

where

```
data Maybe a = Nothing | Just a
```

that is when the unification succeeds it returns `Just` a substitution list of the substitutions which make the terms equal or, when the unification fails, `Nothing`.

(a) Write a function to implement the occurs check:

```
check:: Int -> Term -> Maybe Term
```

(b) Write a function to implement a substitution:

```
subst:: (Int,Term) -> Term -> Term
```

(c) Write a matching function which given two terms returns either `Just` the list of substitutions which results from matching or `Nothing`:

```
match:: Term -> Term -> Maybe[(Int,Term)]
```

(d) Complete the unification algorithm:

```
unify:: Term -> Term -> Maybe[(Int,Term)]
```

15 marks

2. Haskell has two related programming features: list comprehension and monads.

   (a) Given a function `divide: Int -> Int -> Bool` write a list comprehension to find all the divisors of a number:

   ```
   divisors:: Int -> [Int]
   ```

   (b) Describe how a list comprehension is translated into the "core" syntax of Haskell.

   (c) We may modify the `Term` datatype, above, and introduce a monad:

   ```
   data Eq a => Term a = Var a | Opn String [Term a]

   instance Monad Term where
       return x = Var x
       (Var x) >>= q = q x
       (Opn s xs) >>= q = Opn s (map (\y -> y >>= q) xs)
   ```

   Explain what the following program does and show how the code is translated to the "core" syntax of Haskell.

   ```
   subber:: Eq a => a -> Term a -> Term a -> Term a
   subber (x,s) t
      do y <- t
         if x == y then s else return x
   ```

30 marks

3. (a) Determine whether the following two λ-calculus terms are equal (with respect to α, β equality):

$$\lambda xy.((\lambda f.fyx)(\lambda y.yy)) \quad \text{and} \quad (\lambda xy.x)(\lambda x.xx)((\lambda xx.x)(\lambda xx.x))$$

Explain carefully what facts you are using in this determination.

(b) What is meant by saying that $\mathbf{Y}$ is a fixed point combinator? Show that $\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ is a fixed point combinator.

Assuming that one has the arithmetic functions $x - y$, $x < y$, and $x == y$ and the usual encoding of the conditional operator as

$$\text{if} \;\; t \;\; \text{then} \;\; x \;\; \text{else} \;\; y \equiv t\; x\; y$$

show how one can use the fixed point to encode the recursive program:

```
gcd n m = if n == m then  n
          else (if  n < m then gcd n (m - n)
                else gcd m (n - m))
```

(c) How can we represent the term datatype (which I have modified again) in the λ-calculus:

```
data Term a b = Var a | Opn b (Term a b)
```

Provide the definitions of `Var` and `Opn`.

(d) Which of the following are true? If they are not true provide a counter-example:

- In the untyped λ-calculus every λ-term has a normal form.
- In the simply typed λ-calculus every λ-term has a normal form.
- If a term in the simply typed λ-calculus has a normal form then no matter what order you apply β-reductions you will reach the normal form.
- Every program between the natural numbers which can be expressed in the untyped λ-calculus can be also be expressed in the extended (to include natural numbers) simply typed λ-calculus.

$$\frac{x : P \vdash x : P}{x : P, z : \Gamma \vdash x : P}\ \text{proj}$$

$$\frac{x : P, z : \Gamma \vdash t : Q}{z : \Gamma \vdash \lambda x.t : P \to Q}\ \text{abst}$$

$$\frac{z : \Gamma \vdash f : P \to Q \quad z : \Gamma \vdash t : P}{z : \Gamma \vdash (ft) : Q}\ \text{app}$$

Table 1: Annotate type system

25 marks

4. (a) Given the annotated type system in table 1 give the lambda term which corresponds to the following proof:

$$\dfrac{\dfrac{\dfrac{R \to Q \vdash R \to Q}{P, P \to R, R \to Q \vdash R \to Q}\ \text{proj} \qquad \dfrac{\dfrac{P \vdash P}{P, P \to R, R \to Q \vdash P}\ \text{proj} \quad \dfrac{P \to R \vdash P \to R}{P, P \to R, R \to Q \vdash P \to R}\ \text{proj}}{P, P \to R, R \to Q \vdash R}\ \text{app}}{P, P \to R, R \to Q \vdash Q}\ \text{app}}{P \to R, R \to Q \vdash P \to Q}\ \text{abst}$$

(b) Using the type inference system in table 2:

i. Show that the term, $(\lambda x.xx)$, cannot be typed in the simply typed lambda calculus.

ii. Show that $\lambda xy.x(xy)$ can be typed in the simply typed lambda calculus and provide its type.

$$\frac{x : P \vdash x : Q}{x : P, z : \Gamma \vdash x : Q}\; P = Q$$

$$\frac{x : P, z : \Gamma \vdash t : Q}{z : \Gamma \vdash \lambda x.t : R}\; R = P \to Q$$

$$\frac{z : \Gamma \vdash f : P \to Q \quad z : \Gamma \vdash t : P}{z : \Gamma \vdash (f t) : Q}\; \text{fresh } P$$

Table 2: Annotate type system with type equations