COMPUTER SCIENCE 520 (FALL TERM, 2004)
PRINCIPLES OF PROGAMMING LANGUAGES

Solutions for Mid-Term Examination

OCTOBER 29, 2004

Problem 1. ***Recursive Definitions*** *(15 points).* Using "`fix`", the factorial function can be defined as follows:

```
fact = fix (lambda f: Nat -> Nat.
                 lambda x: Nat.
                 if (iszero x) then 1 else (times x (f (pred x))) );
```

We assume that "`times`" was already defined before and correctly implements arithmetical multiplication.

1. Carefully carry out the call-by-value evaluation, step by step, of each of the following applications:

```
fact 0 =

fix (lambda f.lambda x.if (iszero x) then 1 else (times x (f (pred x)))) 0 =>

(lambda x.if (iszero x) then 1 else (times x (fact (pred x)))) 0 =>

if (iszero 0) then 1 else (times 0 (fact (pred 0))) =>

if true then 1 else (times 0 (fact (pred 0))) =>

1
```

Consider the following adjustment of the definition of the factorial above:

```
foo = lambda x: Nat.
        fix(lambda y: Nat.
            if (iszero x) then 1 else (times x (pred y)) );
```

2. Does the definition of "`foo`" type-check? Justify your answer carefully.

Yes. With `x:Nat` and `y:Nat`, we have

`if (iszero x) then 1 else (times x (pred y)):  Nat,`

so that

`lambda y:Nat.if (iszero x) then 1 else (times x (pred y)):  Nat->Nat,`

to which we can apply `fix` according to rule (T-FIX) in [Pierce, p. 144].

3. Ignore the type annotations in the definition of "`foo`", i.e., we don't now care whether "`foo`" type-checks or not. Carefully carry out the call-by-value of each of the following applications:


```
Let t0 = fix (lambda y. if (iszero 0) then 1 else (times 0 (pred y))) and

    t1 = fix (lambda y. if (iszero 1) then 1 else (times 1 (pred y))).


foo 0 =

(lambda x. fix (lambda y. if (iszero x) then 1 else (times x (pred y)))) 0 =>

fix (lambda y. if (iszero 0) then 1 else (times 0 (pred y))) = t0 =>

if (iszero 0) then 1 else (times 0 (pred t0)) =>

if true then 1 else (times 0 (pred t0)) =>

1


foo 1 =

(lambda x. fix (lambda y. if (iszero x) then 1 else (times x (pred y)))) 1 =>

fix (lambda y. if (iszero 1) then 1 else (times 1 (pred y))) = t1 =>

if (iszero 1) then 1 else (times 1 (pred t1)) =>

if false then 1 else (times 1 (pred t1)) =>

times 1 (pred t1) =

times 1 (pred (if (iszero 1) then 1 else (times 1 (pred t1)))) =>

times 1 (pred (if false then 1 else (times 1 (pred t1)))) =>

times 1 (pred (times 1 (pred t1))) =>

...

(diverges)
```

Problem 2. ***Sequencing*** *(15 points).* [Pierce] gives two different presentations of sequencing. The first makes ";" *syntactic sugar*, i.e.,

$$t_1\,;\,t_2 \quad \triangleq \quad (\lambda x : \texttt{Unit}.\, t_2\,)\; t_1 \qquad \text{where } x \notin FV(t_2)$$

The second presentation makes ";" a *constructor*, i.e., $t_1\,;\,t_2$ is an additional form in the syntax of terms, which requires additional rules, two for evaluation and one for typing:

$$\frac{t_1 \Rightarrow t_1'}{t_1\,;\,t_2 \Rightarrow t_1'\,;\,t_2} \qquad \text{(E-SEQ)}$$

$$\texttt{unit}\,;\,t \Rightarrow t \qquad \text{(E-SEQNEXT)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Unit} \qquad \Gamma \vdash t_1 : T}{\Gamma \vdash t_1\,;\,t_2 : T} \qquad \text{(T-SEQ)}$$

A third presentation of ";" (omitted by [Pierce]) makes it a *primitive operator*, in which case ";" is a valid stand-alone term. In this problem we restrict attention to this third presentation. As a primitive operator, it is a bit easier to reason about ";" when written in prefix position, i.e., when written as "$(;\{t_1, t_2\})$", meaning that ";" is applied to the pair "$\{t_1, t_2\}$". But writing ";" in prefix is counter-intuitive, so we still write "$t_1\,;\,t_2$" but now understood as an abbreviation for "$(;\{t_1, t_2\})$".

1. As a primitive operator, ";" takes two terms t_1 and t_2 and returns a new term $t_1\,;\,t_2$. The type of ";" must therefore be of the form $T_1 \times T_2 \rightarrow T_3$. What concrete types will you choose for T_1, T_2 and T_3? Can you choose the same concrete types for all possible situations?

It is not possible to choose the same concrete types for all possible "$t_1\,;\,t_2$". The only requirement is that $T_1 = \texttt{Unit}$ (which is the same for all cases) and $T_2 = T_3$? (which can vary from one case to another). So, for each type T we have a primitive operator ";" of type $\texttt{Unit} \times T \rightarrow T$. This can be made explicit by writing "$;_T$" instead of ";".

2. What is the evaluation rule (or rules) for ";" as a primitive operator?

There are two evaluation rules, identical to the rules used when ";" is a constructor, i.e., rules (E-SEQ) and (E-SEQNEXT).

3. What is the typing rule (or rules) for ";" as a primitive operator?

There is one new typing rule, namely:

$$\frac{}{\Gamma \vdash ;_T\; : \texttt{Unit} \times T \rightarrow T}$$

Problem 3. ***Normalization*** *(10 points).* The statement of the *Normalization Theorem* in [Pierce] is as follows:

If $\vdash t : T$ is derivable in the simply-typable lambda-calculus, i.e., t is a simply-typable closed term of the pure lambda-calculus, then t is normalizable.

1. Why is the Normalization Theorem in [Pierce] restricted to closed terms?

Normalization means termination *at a value*, not just termination. Open terms that cannot be evaluated further, such as $(\lambda \texttt{x}.\,\texttt{x}\,)\texttt{y}$, are *stuck* rather than normalizable or in normal form.

2. Why is the conclusion of the Normalization Theorem in [Pierce] "t is normalizable" rather than "t is strongly normalizable"?

For call-by-value, which is a deterministic evaluation strategy, normalization = strong normalization.

Problem 4. ***Aliasing and Shared State*** *(10 points).*

1. Define the notion of two variables that are *aliases* of each other.

Two reference variables `a` and `b`, i.e., both of type `Ref` T for some type T, are *aliases* if they point to the same cell (or memory location).

2. Consider expressions `(a := 1; a := !b)` and `(a := !b)`. State a condition when both have the same effect, and another condition when both do not have the same effect.

If `a` and `b` are aliases pointing to the same location ℓ such that $\mu(\ell) = 0$, say, then evaluation of `(a := 1; a := !b)` produces a new store $\mu(\ell) = 1$ while evaluation of `(a := !b)` does not change the store. In this case, the two expressions do *not* have the same effect.

If `a` and `b` are *not* aliases, with `a` pointing to ℓ_1 and `b` pointing to ℓ_2 and $\mu(\ell_1) = \mu(ell_2) = 0$, say, then evaluation of both `(a := 1; a := !b)` and `(a := !b)` returns the same store $\mu(\ell_1) = \mu(ell_2) = 0$. In this case, the two expressions have the same effect.

Problem 5. ***Preservation*** *(5 points).* The statement of the *Preservation Theorem* in [Pierce], in the presence of references and side-effects, is as follows:

If $\Gamma \,|\, \Sigma \vdash t : T$ and $\Gamma \,|\, \Sigma \vdash \mu$ and $t \,|\, \mu \Rightarrow t' \,|\, \mu'$, then for some $\Sigma' \supseteq \Sigma$ it holds that $\Gamma \,|\, \Sigma' \vdash t' : T$ and $\Gamma \,|\, \Sigma' \vdash \mu'$.

Why can we not take $\Sigma = \Sigma'$ in the Preservation Theorem above?

Because one step in the evaluation of a term t, e.g., using the rule (E-REFV) in [Pierce, page 166], may have the side-effect of allocating a new location ℓ, i.e., producing a store typing Σ' such that $dom(\Sigma') = dom(\Sigma) \cup \{\ell\}$.

Problem 6. ***Bottom Type and Subtyping*** *(5 points).* In [Pierce] there is a discussion of the "bottom" type, denoted `Bot`. In the syntax of types, we take `Bot` as a primitive base type (just as `Nat`, `Bool`, `Top`, ...). The subtyping rule for `Bot` is:

$$\texttt{Bot} <: T \qquad \text{(S-BOT)}$$

Give a precise argument showing that the type `Bot` is not inhabited, i.e., there is no value (or no well-typed closed term) which has type `Bot`.

Suppose v is a well-typed value, which is not a numerical constant (e.g., 0) nor a boolean constant (e.g., `true`). (The argument to follow can be repeated when v is a constant of the language.) Then v must be an abstraction of the form $\lambda \texttt{x} : T_1 . \, t$ for some T_1 and t. Hence the type assigned to v by a typing derivation must be of the form $T_1 \to T_2$ for some type T_2. Suppose now that $\vdash v : \texttt{Bot}$ is a derivable judgement, i.e., `Bot` is inhabited by v, in order to get a contradiction. By the subsumption rule (T-SUB) in [Pierce, p 186], we also have $\vdash v : T$ for every type T. In particular, we can take $T = \{\,\}$, which is a type that can be derived only for records, not an abstraction. This contradicts that v must be an abstraction. Hence, `Bot` cannot be inhabited.

Problem 7. ***Subtyping Relation*** *(15 points).* For each of the 8 pairs of types below, T and U, determine which of the 4 answers $\{\text{A}, \text{B}, \text{C}, \text{D}\}$ holds.

A. $T <: U$ but not $U <: T$.

B. $U <: T$ but not $T <: U$.

C. Both $T <: U$ and $U <: T$.

D. Neither $T <: U$ nor $U <: T$.

In each case, there is exactly one correct answer. Assume $\texttt{Nat} <: \texttt{Real}$.

1. $T = \texttt{Nat} \to \texttt{Bool}$ and $U = \texttt{Nat} \times \texttt{Bool}$.
   Answer: D
2. $T = (\texttt{Nat} \to \texttt{Nat}) \to \texttt{Nat}$ and $U = (\texttt{Real} \to \texttt{Nat}) \to \texttt{Real}$.
   Answer: A
3. $T = \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}$ and $U = \texttt{Real} \to \texttt{Nat} \to \texttt{Real}$.
   Answer: D
4. $T = \{\texttt{a} : \texttt{Nat}, \texttt{b} : \texttt{Bool}, \texttt{c} : \texttt{Nat}\}$ and $U = \{\texttt{c} : \texttt{Nat}, \texttt{a} : \texttt{Nat}, \texttt{b} : \texttt{Bool}\}$.
   Answer: C
5. $T = \{\texttt{a} : \texttt{Nat}, \texttt{b} : \texttt{Bool}\}$ and $U = \{\texttt{c} : \texttt{Nat}, \texttt{d} : \texttt{Bool}\}$.
   Answer: D
6. $T = \{\,\}$ and $U = \{\texttt{a} : \texttt{Top}\}$.
   Answer: B
7. $T = \{\,\}$ and $U = \texttt{Top}$.
   Answer: A
8. $T = \texttt{Top}$ and $U = \{\texttt{a} : \texttt{Top}\}$.
   Answer: B