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

Partial Solutions for End-of-Term Examination

DECEMBER 20, 2004

Problem 1. ***Sequencing (12 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:

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

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

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

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. **Moreover, we consider the typing of ";" as a primitive operator in the presence of universal types. *Caution*: This is different from a similar problem in the mid-term exam, which was restricted to simple types.** The treatment of ";" in [Pierce], whether as syntactic sugar or as a constructor, is limited to simple types.

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 "$(;\{t_1, t_2\})$".

(4 pts) As a primitive operator, ";" takes two terms t_1 and t_2 and returns a new term $t_1 \,;\, t_2$. The type of each instance of ";" must therefore be of the form $T_1 \times T_2 \to T_3$. Can you assign a single universal type to ";" as a primitive operator? If yes, what is this type? If no, explain carefully.

> Yes, it is possible to assign a single universal type to ";" as a primitive operator, namely, $\forall \texttt{X}.\texttt{Unit} \times \texttt{X} \to \texttt{X}$. Each occurrence of ";" in the program will have a concrete type which is an instance of this universal type, e.g., $\texttt{Unit} \times \texttt{Nat} \to \texttt{Nat}$, $\texttt{Unit} \times \texttt{Bool} \to \texttt{Bool}$, etc.

(4 pts) 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).

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

> There is one new typing rule, namely:
>
> $$\frac{}{\Gamma \vdash \,;\, : \forall \texttt{X}.\texttt{Unit} \times \texttt{X} \to \texttt{X}}$$
>
> Each occurrence of ";" will be given a particular concrete type by using the rule (T-INST), which is already available in the system of universal types.

Problem 2. ***Type Reconstruction and Unification (18 points).*** We consider typability of terms of the pure lambda-calculus in the system of simple types, denoted $\boldsymbol{\lambda}_{\to}$ as usual.

(6 pts) Write the set of constraints, call it C_1, corresponding to the typability of the term $(\lambda \texttt{x}.\, \texttt{x}\,\texttt{x}\,)$ in $\boldsymbol{\lambda}_{\to}$.

> $C_1 = \{\texttt{X} = \texttt{X} \to \texttt{A}\}$ where X corresponds to (the unknown type of) the term variable x.

Does C_1 have a solution? If yes, write down the solution. If no, justify your answer carefully.

> No. C_1 has no solution, i.e., there is no substitution S such that $S(\texttt{X}) = S(\texttt{X} \to \texttt{A})$.

(6 pts) Write the set of constraints, call it C_2, corresponding to the typability of the term $(\lambda \texttt{x}.\, \lambda \texttt{y}.\, \texttt{x}\,(\texttt{x}\,\texttt{y})\,)\,(\lambda \texttt{z}.\,\texttt{z}\,)$ in $\boldsymbol{\lambda}_{\to}$.

> $C_2 = \{\texttt{X} = \texttt{Y} \to \texttt{A}, \texttt{X} = \texttt{A} \to \texttt{B}, \texttt{X} = \texttt{Z} \to \texttt{Z}\}$ where X, Y and Z correspond to (the unknown types of) the term variables x, y and z, respectively.

Does C_2 have a solution? If yes, write down the solution. If no, justify your answer carefully.

> Yes. A most general solution for C_2 is the substitution $S = \{\texttt{A} \mapsto \texttt{Z}, \texttt{B} \mapsto \texttt{Z}, \texttt{X} \mapsto \texttt{Z} \to \texttt{Z}, \texttt{Y} \mapsto \texttt{Z}\}$.

(6 pts) Write the set of constraints, call it C_3, corresponding to the typability of the term $(\lambda \texttt{x}.\, \lambda \texttt{y}.\, \texttt{x}\,(\texttt{x}\,\texttt{y})\,)\,(\lambda \texttt{x}'.\, \lambda \texttt{y}'.\, \texttt{x}'\,)$ in $\boldsymbol{\lambda}_{\to}$.

> $C_3 = \{\texttt{X} = \texttt{Y} \to \texttt{A}, \texttt{X} = \texttt{A} \to \texttt{B}, \texttt{X} = \texttt{X}' \to \texttt{Y}' \to \texttt{X}'\}$ where X, Y, X′ and Y′ correspond to (the unknown types of) the term variables x, y, x′ and y′, respectively.

Does C_3 have a solution? If yes, write down the solution. If no, justify your answer carefully.

> No. C_3 does not have a solution. You can run the unification algorithm on C_3 to find out, but it is easier to examine C_3 directly, as follows. The first two constraints, without the third, have a solution that makes A, B and Y equal to the same type variable (say A) so that $\texttt{X} = \texttt{A} \to \texttt{A}$. But there is no solution for the latter constraint together with the third constraint in C_3, i.e., $\{\texttt{X} = \texttt{A} \to \texttt{A}, \texttt{X} = \texttt{X}' \to \texttt{Y}' \to \texttt{X}'\}$ has no solution, because these two constraints imply $\texttt{A} = \texttt{X}'$ and $\texttt{A} = \texttt{X}' \to \texttt{Y}'$ which cannot be simultaneously solved.

Problem 3. ***Simple Types vs. Universal Types (8 points).*** Justify your answers carefully, in at most 3 lines. If your answer is "yes", provide an example of such a term, possibly with an explanation if you so desire. If your answer is "no", provide a careful explanation.

(4 pts) Is there a term of the pure lambda-calculus which is typable in the system of universal types, but not in the system of simple types.

> Yes. The term $\lambda \texttt{x}.\,\texttt{x}\,\texttt{x}$ is typable in the system of universal types, but not in the system of simple types.

(4 pts) Is there a term of the pure lambda-calculus which is typable in the system of simple types, but not in the system of universal types.

> No. The system of simple types is a subsystem of the system of universal types, i.e., every typing derivation in the first is a legal typing derivation in the second.

Problem 4. ***Positive Recursive Types vs. Recursive Types (8 points).*** Justify your answers carefully, in at most 3 lines. If your answer is "yes", provide an example of such a term, possibly with an explanation if you so desire. If your answer is "no", provide a careful explanation.

(4 pts) Is there a term of the pure lambda-calculus which is typable in the system of recursive types, but not in the system of positive recursive types.

> Yes. Every term of the pure lambda-calculus is typable in the system of recursive types, in particular, every term which is not strongly normalizable, such as $(\lambda \mathtt{x}.\, \mathtt{x}\,\mathtt{x})\,(\lambda \mathtt{x}.\, \mathtt{x}\,\mathtt{x})$. On the other hand, every term of the pure lambda-calculus which is typable in the system of positive recursive types is strongly normalizable, which implies $(\lambda \mathtt{x}.\, \mathtt{x}\,\mathtt{x})\,(\lambda \mathtt{x}.\, \mathtt{x}\,\mathtt{x})$ is not typable in it.

(4 pts) Is there a term of the pure lambda-calculus which is typable in the system of positive recursive types, but not in the system of recursive types.

> No. The system of positive recursive types is a subsystem of the system of universal types. Alternatively, not all terms of the pure lambda-calculus are typable in the system of positive recursive types, whereas all terms of the pure lambda-calculus are typable in the system of recursive types.

Problem 5. ***Congruence Relations (20 points).*** In lecture and lecture notes we defined $\approx$ as the least congruence on recursive types satisfying two axioms:

1. $\boxed{\mu X.T \;\approx\; \mu Y.[X \mapsto Y]T}$
   i.e., if two recursive types are the same up to renaming of bound variables then the two are congruent.
2. $\boxed{\mu X.T \;\approx\; [X \mapsto \mu X.T]T}$
   i.e., if a recursive type (on the right of $\approx$) is obtained from another recursive type (on the left of $\approx$) by "unwinding" once the loop corresponding to the μ-binding, then the two are congruent.

We want to extend the congruence $\approx$ in three different ways. In each case we state the congruence in English, together with an example, and we want you to write the corresponding axiom formally (in the style of the grayboxes above). If you wish, you may add a short explanation.

3. If a recursive type is obtained from another recursive type by permuting two adjacent μ-bindings, then the two types are congruent. For example, we want to be able to derive:

   $\mu \mathtt{X}.\mu \mathtt{Y}.\mathtt{X} \to (\mathtt{Nat} \times \mathtt{Y}) \approx \mu \mathtt{Y}.\mu \mathtt{X}.\mathtt{X} \to (\mathtt{Nat} \times \mathtt{Y})$

   (2 pts) $\boxed{\mu X.\mu Y.T \;\approx\; \mu Y.\mu X.T}$

4. If a recursive type is obtained from another recursive type by merging two adjacent μ-bindings, then the two types are congruent. For example, we want to be able to derive:

   $\mu \mathtt{X}.\mu \mathtt{Y}.\, \mathtt{X} \to (\mathtt{Nat} \times \mathtt{Y}) \approx \mu \mathtt{X}.\, \mathtt{X} \to (\mathtt{Nat} \times \mathtt{X})$

   (2 pts) $\boxed{\mu X.\mu Y.T \;\approx\; \mu X.[Y \mapsto X]T}$

5. If a recursive type is obtained from another recursive type by permuting the two components of a product, then the two types are congruent. For example, we want to be able to derive:

   $\mathtt{Nat} \times \mathtt{Bool} \approx \mathtt{Bool} \times \mathtt{Nat}$

   (2 pts) $\boxed{T_1 \times T_2 \;\approx\; T_2 \times T_1}$

In addition to the preceding 5 axioms, we consider an additional one:

6. $\boxed{\mu X.T_1 \to T_1 \to T_2 \;\approx\; T_1 \to \mu X.T_1 \to T_2 \qquad \text{where } X \notin FTV(T_1)}$

Let A be a subset of the 6 axioms above, i.e., $A \subseteq \{1,2,3,4,5,6\}$. For arbitrary recursive types T_1 and T_2, we write $T_1 {\approx_A} T_2$ iff there is a derivation to show that T_1 is congruent to T_2 using only axioms in A. Thus, $\approx_{\{1,2\}}$ is the same as $\approx$ presented in lecture and in lecture notes.

If T is a recursive type, let $\lceil T \rceil$ denote the possibly infinite type (or parse tree) obtained by repeatedly unwinding all the μ-bindings in T. We say that a set A of axioms is *sound* if $T_1 {\approx_A} T_2$ implies $\lceil T_1 \rceil = \lceil T_2 \rceil$.

(4 pts) Show that the set of axioms $A = \{1,2,6\}$ is not sound, justifying your answer in at most 3 lines.

> Consider the recursive type $T \triangleq \mu \mathtt{X}.\mathtt{Nat} \to \mathtt{Nat} \to \mathtt{X} \to \mathtt{Nat}$ which, according to axiom 6, is congruent to the recursive type $T' \triangleq \mathtt{Nat} \to \mu \mathtt{X}.\mathtt{Nat} \to \mathtt{X} \to \mathtt{Nat}$. But $\lceil T \rceil \neq \lceil T' \rceil$.

(10 pts) Determine which of the following sets of axioms is sound, justifying your answer in at most 3 lines:

(4 pts) $A = \{1,2,3\}$

> Sound. Permuting adjacent μ bindings does not affect the infinite type obtained by repeatedly unwinding them.

(4 pts) $A = \{1,2,4\}$

> Sound. Merging adjacent μ bindings does not affect the infinite type obtained by repeatedly unwinding them.

(2 pts) $A = \{1,2,5\}$

> Unsound. (Explanation omitted.)

Problem 6. ***Recursive Types and Subtyping (16 points).*** Determine which of the following is true and which is false. In each case, add an explanation not exceeding 3 lines. As usual, we take $\mathtt{Nat} <: \mathtt{Real}$ and $\mathtt{Real} \not<: \mathtt{Nat}$

(4 pts) $\mu \mathtt{X}.\mathtt{Real} \to (\mathtt{Nat} \times \mathtt{X}) \;<:\; \mu \mathtt{X}.\mathtt{Nat} \to (\mathtt{Real} \times \mathtt{X})$

> True. (Explanation omitted.)

(4 pts) $\mu \mathtt{X}.\mathtt{X} \to (\mathtt{Nat} \times \mathtt{X}) \;<:\; \mu \mathtt{X}.\mathtt{X} \to (\mathtt{Real} \times \mathtt{X})$

> False. (Explanation omitted.)

(4 pts) $\mu \mathtt{X}.\mu \mathtt{Y}.\mathtt{Y} \to (\mathtt{Nat} \times \mathtt{X}) \;<:\; \mu \mathtt{X}.\mu \mathtt{Y}.\mathtt{Y} \to (\mathtt{Real} \times \mathtt{X})$

> False. (Explanation omitted.)

(4 pts) $\mu \mathtt{X}.(\mathtt{X} \to \mathtt{Nat}) \to \mathtt{Nat} \;<:\; \mu \mathtt{X}.\mathtt{X} \to \mathtt{Real}$

> False. (Explanation omitted.)

This last part was made easier due to an inadvertent typographical mistake. My intention was to write "$\mu \mathtt{X}.(\mathtt{X} \times \mathtt{Nat}) \times \mathtt{Nat} \;<:\; \mu \mathtt{X}.\mathtt{X} \times \mathtt{Real}$", not "$\mu \mathtt{X}.(\mathtt{X} \to \mathtt{Nat}) \to \mathtt{Nat} \;<:\; \mu \mathtt{X}.\mathtt{X} \to \mathtt{Real}$", in which case the correct answer is "true" or "false" depending on how you define the congruence $\approx$.