COMPUTER SCIENCE 520 (FALL TERM, 2006)
PRINCIPLES OF PROGAMMING LANGUAGES

Solutions for Mid-Term Examination

OCTOBER 26, 2006

Problem 1. ***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 $T_1 \times T_2 \to T_3$. 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 $\texttt{Unit} \times T \to T$. 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:
>
> $$\frac{}{\Gamma \vdash \;;_T\; : \texttt{Unit} \times T \to T}$$

Problem 2. ***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 $(\lambda \mathtt{x}.\, \mathtt{x}\,)\mathtt{y}$, 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 3. ***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 4. ***Preservation*** *(15 points).* The statement of the *Preservation Theorem* in [Pierce] for the simply-typed lambda-calculus (STLC) is as follows:

> If $\Gamma \vdash t : T$ and $t \Rightarrow t'$ then $\Gamma \vdash t' : T$.

The converse of the Preservation Theorem for STLC is:

> If $t \Rightarrow t'$ and $\Gamma \vdash t' : T$ then $\Gamma \vdash t : T$.

1. Give an example showing that the converse of the Preservation Theorem for STLC does not hold in general.

Consider the term $t = ((\lambda x.\, \lambda y.\, y\ )\ \Omega)$ where $\Omega = (\lambda z.\, z\ z\ )\ (\lambda z.\, z\ z\ )$, and the term $t' = \lambda y.\, y$. Then $t \Rightarrow t'$. While t' is typable in STLC, t is not.

2. State a restriction on STLC so that the converse of the Preservation Theorem holds.

Restrict STLC to terms of the lambda-calculus where every binding occurrence of the form "λx" binds exactly one bound occurrence of variable x.

The statement of the *Preservation Theorem* in [Pierce] for the simply-typed lambda-calculus augmented with references and side-effects (STLC+ref) is as follows:

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

3. Why can we not take $\Sigma = \Sigma'$ in the Preservation Theorem for STLC+ref?

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 5. ***Sum Types and Conditionals*** *(15 points).* (For a reminder: See Handout 09, pages 11, 14 and 15.) We want to show that in the presence of "`Unit`" and sum types, we can eliminate "`Bool`" as a base type (and therefore "`true`", "`false`" and "`if - then - else`" as primitives). For this, we first make "`Bool`" syntactic sugar, i.e., we define it using the remaining primitives (in the syntax of types). Specifically, we define

$$\texttt{Bool} \triangleq \texttt{Unit} + \texttt{Unit}$$

1. Given the preceding definition of "`Bool`", define appropriately "`true`" and "`false`" using other primitives (in the syntax of terms).

Define $\texttt{true} \triangleq (\texttt{inl unit})$ and $\texttt{false} \triangleq (\texttt{inr unit})$. As just defined, both `true` and `false` have type $\texttt{Bool} \triangleq \texttt{Unit} + \texttt{Unit}$.

2. Given the preceding definition of "`Bool`", "`true`" and "`false`", define appropriately "`if - then - else`" using other primitives (in the syntax of terms).

Define $(\texttt{if } t_0 \texttt{ then } t_1 \texttt{ else } t_2) \triangleq (\texttt{case } t_0 \texttt{ of inl } x_1 \Rightarrow t_1 \mid \texttt{inr } x_2 \Rightarrow t_2)$.

3. Verify your definitions, i.e., show that for arbitrary terms t_1 and t_2:

$$\texttt{if true then } t_1 \texttt{ else } t_2 \quad \Rightarrow^* t_1$$
$$\texttt{if false then } t_1 \texttt{ else } t_2 \Rightarrow^* t_2$$

after "`if - then - else`", "`true`" and "`false`" are replaced by their definitions.

Substitute $\texttt{true} \triangleq (\texttt{inl unit})$ for first argument in the definition of "`if - then - else`" to get:

$$\texttt{case } (\texttt{inl unit}) \texttt{ of inl } x_1 \Rightarrow t_1 \mid \texttt{inr } x_2 \Rightarrow t_2$$

which, according to the evaluation rule in [Pierce, page 132], is evaluated to

$$[x_1 \mapsto \texttt{unit}]t_1 = t_1$$

where the equality holds because x_1 does not occur in t_1. For the other case, when we substitute `false` for the first argument in the definition of "`if - then - else`", the verification is similar.

Problem 6. ***Evaluation Contexts*** *(15 points).* In Handouts 14 and 15 we discussed evaluation contexts relative to a simple foundational language $\mathcal{L}$. We now augment $\mathcal{L}$ with conditionals and boolean values, to define a new language $\mathcal{L} + \mathsf{cond}$, as follows:

<u>SYNTAX OF $\mathcal{L} + \mathsf{cond}$</u>

t	$::=$		*terms*
		nf	*normal form*
		$t\ t$	*application*
		$t + t$	*summation*
		$\mathtt{eq}\ t\ t$	*equality of numerals*
		$\mathtt{if}\ t\ \mathtt{then}\ t\ \mathtt{else}\ t$	*conditional*
nf	$::=$		*normal forms*
		x	*variable*
		v	*value*
v	$::=$		*values*
		b	*boolean* $(\mathtt{true}, \mathtt{false})$
		n	*decimal numeral* $(0, 1, 2, \ldots)$
		$\lambda x.\, t$	*abstraction*

We wish to specify the operational semantics for $\mathcal{L} + \mathsf{cond}$ using evaluation contexts. Recall that for the original $\mathcal{L}$, the *call-by-value and left-to-right evaluation* was specified as follows:

<u>SEMANTICS OF $\mathcal{L}$</u>

$$E ::= [\,] \mid E\ t \mid v\ E \mid E + t \mid v + E$$

(a) $E[(\lambda x.\, t\,)\ v] \rightsquigarrow E[\,[x \mapsto v]t\,]$

(b) $E[n + n'] \rightsquigarrow E[n'']$ where n'' is the sum of n and n'

Write the evaluation contexts and rules for $\mathcal{L} + \mathsf{cond}$ under each of the following strategies:

1. Call-by-value, left-to-right, and every conditional is evaluated once its guard has been evaluated without waiting for its `then`-branch or `else`-branch to be evaluated.

$$E ::= [\,] \mid E\ t \mid v\ E \mid E + t \mid v + E \mid \mathtt{eq}\, E\ t \mid \mathtt{eq}\, v\ E \mid \mathtt{if}\ E\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2$$

(a) $E[(\lambda x.\, t\,)\ v] \rightsquigarrow E[\,[x \mapsto v]t\,]$

(b) $E[n + n'] \rightsquigarrow E[n'']$ where n'' is the sum of n and n'

(c) $E[\mathtt{eq}\, n\, n\,] \rightsquigarrow E[\mathtt{true}]$

(d) $E[\mathtt{eq}\, n\, n'\,] \rightsquigarrow E[\mathtt{false}]$ where n and n' are distinct

(e) $E[\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2] \rightsquigarrow E[t_1]$

(f) $E[\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2] \rightsquigarrow E[t_2]$

2. Call-by-value, left-to-right, and every conditional is evaluated only after all three of its arguments (guard, `then`-branch, and `else`-branch) have been evaluated.

$$E ::= [\,] \mid E\ t \mid v\ E \mid E + t \mid v + E \mid \texttt{eq}\, E\, t \mid \texttt{eq}\, v\, E \mid \texttt{if}\ E\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \mid \texttt{if}\ v\ \texttt{then}\ E\ \texttt{else}\ t \mid \texttt{if}\ v_1\ \texttt{then}\ v_2\ \texttt{else}\ E$$

(a) $E[(\lambda x.\, t\,)\ v] \rightsquigarrow E[\,[x \mapsto v]t\,]$
(b) $E[n + n'] \rightsquigarrow E[n'']$ where n'' is the sum of n and n'
(c) $E[\texttt{eq}\, n\, n\,] \rightsquigarrow E[\texttt{true}]$
(d) $E[\texttt{eq}\, n\, n'\,] \rightsquigarrow E[\texttt{false}]$ where n and n' are distinct
(e) $E[\texttt{if true then}\ v_1\ \texttt{else}\ v_2] \rightsquigarrow E[v_1]$
(f) $E[\texttt{if false then}\ v_1\ \texttt{else}\ v_2] \rightsquigarrow E[v_2]$

3. Call-by-name, left-to-right, and every conditional is evaluated once its guard has been evaluated without waiting for its `then`-branch and `else`-branch to be evaluated. (Same as 1 above, except that "call-by-value" is replace by "call-by-name".)

$$E ::= [\,] \mid E\ t \mid E + t \mid v + E \mid \texttt{eq}\, E\, t \mid \texttt{eq}\, v\, E \mid \texttt{if}\ E\ \texttt{then}\ t_1\ \texttt{else}\ t_2$$

(a) $E[(\lambda x.\, t\,)\ t'] \rightsquigarrow E[\,[x \mapsto t']t\,]$
(b) $E[n + n'] \rightsquigarrow E[n'']$ where n'' is the sum of n and n'
(c) $E[\texttt{eq}\, n\, n\,] \rightsquigarrow E[\texttt{true}]$
(d) $E[\texttt{eq}\, n\, n'\,] \rightsquigarrow E[\texttt{false}]$ where n and n' are distinct
(e) $E[\texttt{if true then}\ t_1\ \texttt{else}\ t_2] \rightsquigarrow E[t_1]$
(f) $E[\texttt{if false then}\ t_1\ \texttt{else}\ t_2] \rightsquigarrow E[t_2]$

4. Same as 3 above – but now, in addition, evaluation is blocked from the moment an argument of a binary constructor (`eq` or $+$) evaluates to a value which is not a numeral, or the first argument of a conditional evaluates to a value which is not a boolean.

$$E ::= [\,] \mid E\ t \mid E + t \mid n + E \mid \texttt{eq}\, E\, t \mid \texttt{eq}\, n\, E \mid \texttt{if}\ E\ \texttt{then}\ t_1\ \texttt{else}\ t_2$$

(a) $E[(\lambda x.\, t\,)\ t'] \rightsquigarrow E[\,[x \mapsto t']t\,]$
(b) $E[n + n'] \rightsquigarrow E[n'']$ where n'' is the sum of n and n'
(c) $E[\texttt{eq}\, n\, n\,] \rightsquigarrow E[\texttt{true}]$
(d) $E[\texttt{eq}\, n\, n'\,] \rightsquigarrow E[\texttt{false}]$ where n and n' are distinct
(e) $E[\texttt{if true then}\ t_1\ \texttt{else}\ t_2] \rightsquigarrow E[t_1]$
(f) $E[\texttt{if false then}\ t_1\ \texttt{else}\ t_2] \rightsquigarrow E[t_2]$