CMCS 22100/32100 — Programming Languages
Midterm Examination Solution
November 12, 2004

Question 1 Commutativity of substitution (25 points)
Show that for any expressions e, e_1, and e_2 of the Arith+Let language, if $x_1 \notin \mathrm{FV}(e_2)$ and $x_2 \notin \mathrm{FV}(e_1)$, then

$$\{e_1/x_1\}(\{e_2/x_2\}e) = \{e_2/x_2\}(\{e_1/x_1\}e) \tag{1}$$

The abstract syntax of Arith+Let is given by:

$$\begin{aligned} e \ ::= \ & \texttt{num}[n] \mid \texttt{var}(x) \mid \\ & \texttt{plus}(e_1, e_2) \mid \texttt{times}(e_1, e_2) \mid \\ & \texttt{let}(e_1,\, x.e_2) \end{aligned}$$

Solution. We prove (1) by induction on the structure of the term e. Note that we implicitly assume that $x_1 \neq x_2$, since the equation is in fact not in general true if $x_1 = x_2$ (what is a counterexample?).

Case $e = \texttt{num}[n]$. Here $\{e_1/x_1\}e = e$ and $\{e_1/x_2\}e = e$, so

$$\{e_1/x_1\}(\{e_2/x_2\}e) = \{e_1/x_1\}e = e = \{e_2/x_2\}e = \{e_2/x_2\}(\{e_1/x_1\}e)$$

Case $e = \texttt{var}(x)$. If $x \neq x_1$ and $x \neq x_2$, then $\{e_1/x_1\}e = e$ and $\{e_1/x_2\}e = e$, so both sides of (1) are equal to e as in the first case.

If $x = x_1$, then $\{e_1/x_1\}e = e_1$ and $\{e_2/x_2\}e = e$, so

$$\{e_1/x_1\}(\{e_2/x_2\}e) = \{e_1/x_1\}e = e_1 \tag{2}$$

and for the left hand side of (1) we have

$$\{e_2/x_2\}(\{e_1/x_1\}e) = \{e_2/x_2\}e_1 \tag{3}$$

But since x_2 is not free in e_1, we have $\{e_2/x_2\}e_1 = e_1$ by the following Lemma. Thus both sides of (1) are equal to e_1.

> **Lemma** 1.1: If $x \notin FV(e)$ then $\{e'/x\}e = e$.
> **Proof:** This is another straightforward induction on the structure of e similar to, but simpler than, the current proof. It is left as an exercise.

Finally, if $x = x_2$, the reasoning is similar, with both sides being equal to e_2 in this case.

Case $e = \texttt{plus}(e_3, e_4)$. By induction, we have

$$\begin{aligned} \{e_1/x_1\}(\{e_2/x_2\}e_3) &= \{e_2/x_2\}(\{e_1/x_1\}e_3) \qquad (4) \\ \{e_1/x_1\}(\{e_2/x_2\}e_4) &= \{e_2/x_2\}(\{e_1/x_1\}e_4) \qquad (5) \end{aligned}$$

and therefore

$$\begin{array}{rcll}\{e_1/x_1\}(\{e_2/x_2\}\texttt{plus}(e_3,e_4)) & = & \{e_1/x_1\}(\texttt{plus}(\{e_2/x_2\}e_3,\{e_2/x_2\}e_4)) & \\ & = & \texttt{plus}(\{e_1/x_1\}(\{e_2/x_2\}e_3,\{e_1/x_1\}(\{e_2/x_2\}e_4)) & \\ & = & \texttt{plus}(\{e_2/x_2\}(\{e_1/x_1\}e_3,\{e_2/x_2\}(\{e_1/x_1\}e_4)) & \text{by } (4,5) \\ & = & \texttt{plus}(\{e_2/x_2\}(\{e_1/x_1\}e_3,\{e_2/x_2\}(\{e_1/x_1\}e_4)) & \\ & = & \{e_2/x_2\}(\texttt{plus}(\{e_1/x_1\}e_3,\{e_1/x_1\}e_4)) & \\ & = & \{e_2/x_2\}(\{e_1/x_1\}\texttt{plus}(e_3,e_4)) & \end{array}$$

Case $e = \times(e_3, e_4)$. The proof for this case is exactly analagous to that for the previous case, *mutatis mutandis*.

Case $e = \texttt{let}(e_3, y.e_4)$. Here by alpha conversion we can assume that the bound variable y is distinct from x_1 and x_2 and is not free in any of e_1, e_2, e_3. We have

$$\begin{array}{rcl}\{e_1/x_1\}(\{e_2/x_2\}e & = & \{e_1/x_1\}(\texttt{let}(\{e_2/x_2\}e_3,\{e_2/x_2\}(y.e_4))) \\ & = & \texttt{let}(\{e_1/x_1\}(\{e_2/x_2\}e_3),\{e_1/x_1\}(\{e_2/x_2\}(y.e_4))) \\ & = & \texttt{let}(\{e_1/x_1\}(\{e_2/x_2\}e_3),y.(\{e_1/x_1\}(\{e_2/x_2\}e_4)))\end{array} \tag{6}$$

By induction, we have

$$\{e_1/x_1\}(\{e_2/x_2\}e_3) = \{e_2/x_2\}(\{e_1/x_1\}e_3) \tag{7}$$
$$\{e_1/x_1\}(\{e_2/x_2\}e_4) = \{e_2/x_2\}(\{e_1/x_1\}e_4) \tag{8}$$

So we have

$$\begin{array}{l}\texttt{let}(\{e_1/x_1\}(\{e_2/x_2\}e_3),y.(\{e_1/x_1\}(\{e_2/x_2\}e_4))) \\ \quad = \ \texttt{let}(\{e_2/x_2\}(\{e_1/x_1\}e_3),y.(\{e_2/x_2\}(\{e_1/x_1\}e_4))) \quad \text{by } (7,8) \\ \quad = \ \ldots \\ \quad = \ \{e_2/x_2\}(\{e_1/x_1\}\texttt{let}(e_3,y.e_4)\end{array} \tag{9}$$

Finally, equations (6) and (9) imply (1).

Note: The considerable notational clutter of this proof could have been reduced if we had given names to the two substitutions, say $\sigma_1 = \{e_1/x_1\}$ and $\sigma_2 = (\{e_2/x_2\}$. Then (7), for instance, could be more concisely expressed as:

$$\sigma_1(\sigma_2\ e_3) = \sigma_2(\sigma_1\ e_3)$$

Question 2 Call-by-Name MinML (30 points)
The Call-by-Name (CBN) version of MinML differs from the one presented by Harper and discussed in class in one way: in function applications the function arguments are "passed" before they are evaluated, not after evaluation as in the normal, Call-by-Value (CBV), MinML. The primitive operator expressions like $+(e_1, e_2)$ still need to have their arguments evaluated before they can be reduced.

(a) (10 points): Give any new or changed rules for small-step evaluation ($\mapsto$) for the CBN MinML.

Solution: We revise rule (9.16) for CBN semantics.

$$\frac{(v = \texttt{fun}\, f(x : \tau_1) : \tau_2 \;\texttt{is}\; e)}{\texttt{apply}(v, e_1) \mapsto \{v, e_1/f, x\}e} \quad (9.16 - \text{CBN})$$

We also remove the search rule (9.21) from the system because under CBN we no longer need to evaluate the argument before β-reduction (*i.e.*, before rule (9.16-CBN) can be applied).

(b) (5 points): Do the typing rules for CBN MinML differ from those of the normal CBV MinML? If so, show the altered typing rules.

Solution: No, the typing rules do not change. The fact that the argument expression is substituted before rather than after its evaluation does not change the requirement that its type match the domain type of the function.

(c) (15 points): Discuss how the change from CBV to CBN affects the proofs of the Preservation and Progress Theorems (i.e. which cases change, and how?).

Solution:

Preservation: $\Gamma \vdash e : \tau, e \mapsto e' \Rightarrow \Gamma \vdash e' : \tau$.

The case for `apply` changes slightly to reflect the new (9.16-CBN) transition rule: the Substitution Lemma must now be used on $\{v, e_1/f, x\}e$ instead of $\{v, v_1/f, x\}e$.

Progress: $\Gamma \vdash e : \tau \Rightarrow e$ is a value or $\exists e' : e \mapsto e'$.

In the case where $\Gamma \vdash e : \tau$ by (9.12) and the subcase where $e = \texttt{apply}(v_1, e_2)$ and v_1 is a value, $\exists e'. e \mapsto e'$ by (9.16-CBN) regardless of whether e_2 is a value, so we do not have to consider alternate subcases where e_2 is a value or makes a transition.

(d) (Bonus question, 10 points): Give an example of an expression whose evaluation terminates in the CBN semantics, but does not terminate in the CBV semantics.

Solution: We can solve this problem in two steps. First, we need to create an expression that does not terminate under CBV. We do this using an infinite recursion:

$$e = \texttt{apply}((\texttt{fun}\; g\; (y : \texttt{int}) : \texttt{int is apply}(g, y)), \texttt{num}[0])$$

Second, we define a function which does not actually reference its argument variable in its body, and apply it to e. In a CBV regime, we always try to fully evaluate the argument of any function application, so if the argument expression doesn't terminate neither does the function application. On the other hand, in CBN, if the argument doesn't need to be evaluated *after* it has been passed to the function, because the argument variable is not mentioned, the application will converge. So here is an application expression that will diverge under CBV but terminate under CBN:

$$\texttt{apply}((\texttt{fun}\; f\; (x : \texttt{int}) : \texttt{int is num}[1]), e)$$

Question 3 Evaluation with Free Variables (25 points)
The small-step evaluation semantics $\mapsto$ for MinML can be considered to be defined even for open expressions (expressions containing free variables, i.e. expressions e such that $\mathrm{FV}(e) \neq \emptyset$). There are no transitions for variables, so expressions of the form x where x is a variable are stuck. We will assume here that all function expressions are values, even if they are not closed.

(a) (20 points) Prove that for any expressions e and e' (possibly open)

$$e \mapsto e' \Rightarrow \mathrm{FV}(e') \subseteq \mathrm{FV}(e) \tag{10}$$

(*i.e.*, transition steps introduce no new free variables).

Solution: We prove (10) by induction on the rules for deriving $e \mapsto e'$.

Case: $e \mapsto e'$ by Rule (9.13). In this case $e = +(m, n)$ and $e' = p$ where $p = m + n$. Since e' is a number, $FV(e') = \emptyset$, which is a subset of any set, and so $FV(e') \subseteq FV(e)$. The cases for instruction rules for other primitive operators are similar.

Case: $e \mapsto e'$ by Rule (9.14). Here $e = \texttt{if true then } e_1 \texttt{ else } e_2$ and $e' = e_1$, and we have:

$$FV(e') = FV(e_1) \subseteq FV(e_1) \cup FV(e_2) = FV(e)$$

Case: $e \mapsto e'$ by Rule (9.15). Similar to (9.14).

Case: $e \mapsto e'$ by Rule (9.16). So $e = \texttt{apply}(v_1, v_2)$ where $v_1 = \texttt{fun } f(x : \tau_1) : \tau_2 \texttt{ is } e_1$, and $e_1 = \{v_1, v_2/f, x\}e_1$. We have

$$FV(e') = FV(v_1) \cup FV(v_2) \quad \text{where } FV(v_1) = FV(e_1) - \{f, x\} \tag{11}$$

To give a bound on the free variable set for e', we need the following Lemma relating free variables and substitution.

> **Lemma** 3.1: For any term e and substitution σ,
>
> $$FV(\sigma(e)) \subseteq \left(\bigcup_{x \in dom(\sigma)} FV(\sigma(x)) \right) \cup (FV(e) - dom(\sigma)) \tag{12}$$
>
> Where $dom(\sigma)$ is the domain of the substitution σ, *i.e.*, the set of variables that it replaces. The proof is by structural induction on e.

So, by applying Lemma 3.1 to e_1, we have:

$$\begin{aligned} FV(e') &\subseteq FV(v_1) \cup FV(v_2) \cup (FV(e_1) - \{f, x\}) \\ &= FV(v_1) \cup FV(v_2) \\ &= FV(e) \end{aligned}$$

Case: $e \mapsto e'$ by Rule (9.17). Here $e = +(e_1, e_2)$ and $e' = +(e_1', e_2)$ where $e_1 \mapsto e_1'$. By induction, we can assume $FV(e_1') \subseteq FV(e_1)$. Hence

$$\begin{aligned} FV(e') &= FV(e_1') \cup FV(e_2) \\ &\subseteq FV(e_1) \cup FV(e_2) \\ &= FV(e) \end{aligned}$$

The proofs for rules (9.18) through (9.21) are similar to that for (9.17).

(b) (5 points) Give an example of an open expression e such that $e \mapsto^* v$ for some closed value expression v.

Question 4 Eta-equivalence (20 points)
Suppose $\vdash e : t \to t'$ in MinML and define

$$e' = \texttt{fun}\, f(x : t) : t' \;\texttt{is}\; \texttt{apply}(e, x)$$

where f and x are fresh bound variables. The expression e' is called an *eta-expansion* of e in lambda calculus terminology.

Give an informal argument showing that e and e' are equivalent in the sense that for any expression e_0 such that $\vdash e_0 : t$, $\texttt{apply}(e, e_0)$ and $\texttt{apply}(e', e_0)$ evaluate to the same result (i.e. if either evaluates to a value v, then both evaluate to v, and if either fails to terminate, then both fail to terminate).

Solution: We need to show that $\texttt{apply}(e, e_0) \overset{*}{\mapsto} v \Leftrightarrow \texttt{apply}(e', e_0) \overset{*}{\mapsto} v$ and $\texttt{apply}(e, e_0)$ fails to terminate $\Leftrightarrow \texttt{apply}(e', e_0)$ fails to terminate.

Two remarks about common mistakes:

1. Although type safety guarantees that a well-typed term will not be stuck, it *does not* make any guarantees about termination, ***i.e.***, well-typed terms can loop forever. Thus, showing that both terms are well-typed does not prove termination.
2. Remember that the evaluation strategy in MinML is CBV. Thus, in general, function application $(e_1 e_2)$ happens in three stages:
   (a) Evaluate e_1 to a value v_1 (v_1 will be a function if the application is well typed) using zero or more applications of rule (9.20).
   (b) Evaluate e_2 to a value v_2 (v_2 can be any value of the appropriate type) using zero or more applications of rule (9.21).
   (c) Substitute v_2 into the body of v_1 provided that v_1 is indeed a function using rule (9.16).

   We cannot skip any of these stages except where the subterm in question is already a value, as is the case with $e' = \texttt{fun}\; f\; (x : \tau) : \tau' \;\texttt{is}\; \texttt{apply}(e, x)$. Recall that (9.16) does not even fire until both the function *and* the argument are fully evaluated. Since e and e_0 are arbitrary expressions they may need to be evaluated using (9.21) before β-reduction (9.16).

The evaluation of $\texttt{apply}(e', e_0)$ proceeds as follows:

$$\begin{aligned}
\texttt{apply}(e', e_0) &\overset{*}{\mapsto} \texttt{apply}(e', v_0) \quad \text{by applications of (9.21)} && (13)\\
&\mapsto \texttt{apply}(e, v_0) \quad \text{by (9.16)} && (14)\\
&\overset{*}{\mapsto} \texttt{apply}(v, v_0) \quad \text{by applications of (9.20)} && (15)\\
&\mapsto e'' \quad \text{by (9.16)} && (16)\\
&\overset{*}{\mapsto} v_f && (17)
\end{aligned}$$

while the evaluation of $\texttt{apply}(e, e_0)$ goes like this:

$$\begin{aligned}
\texttt{apply}(e, e_0) &\overset{*}{\mapsto} \texttt{apply}(v, e_0) \quad \text{by (9.20)} && (18)\\
&\overset{*}{\mapsto} \texttt{apply}(v, v_0) \quad \text{by (9.21)} && (19)\\
&\mapsto e'' \quad \text{by (9.16)} && (20)\\
&\overset{*}{\mapsto} v_f && (21)
\end{aligned}$$

where e'' is $\{v, v_0/g, y\}e_1$ assuming $v = \texttt{fun}\, g\,(y : \tau_1) : \tau_2 \;\texttt{is}\; e_1$.

Notice that the two expressions are not directly equivalent. In general, it takes an arbitrary number of steps to get to a point of convergence, namely the expression $\texttt{apply}(v, v_0)$. These intermediate sequences of steps may also fail to terminate; every time we use the search rules (9.20) or (9.21) to fully evaluate a subterm we may have nontermination because evaluating that subterm may loop forever. Thus the first reduction sequence may fail to terminate at (13) because evaluation of e_0 does not terminate, or at (15) because evaluation of e does not terminate, or at (17) because e'' fails to terminate. The second reduction will also fail to terminate for the same reasons: at (19) if e_0 does not terminate, at (18) if e does not terminate, or at (21) if e'' does not terminate.

Thus it is clear that $\texttt{apply}(e', e_0)$ and $\texttt{apply}(e, e_0)$ will either both terminate, yielding the same value v_f, or they will both fail to terminate for one of the three reasons cited above.